Part IIA Third Year Projects

Standard Project
VLSI Design

Project SB1

April 2015

Project guide

This Lab guide is supplemented by material on the project's Moodle worksite, **SB1: VLSI Design**, for which all participants will receive a login.

Worksite: `https://www.vle.cam.ac.uk`
Email support: `moodlehelp@uis.cam.ac.uk`

Part IIA Third Year Projects

Standard Project in VLSI Design SB1

Introduction

The aims of this project are to provide a degree of familiarity with the following:

- The potential of computer-aided design for crafting a **System-on-a-Chip**
- The tools available for key activities in electronic system design
- The value of hardware description languages (HDL)
- The importance of hierarchy and design re-use in IC design
- The structure and detailed design of MOS transistors
- Development of simple digital circuits based on MOS transistors
- Numerical simulation of digital systems and MOS transistor circuits
- The importance of parasitic capacitance in determining operating speed
- The relationship between simulated results and measured performance

We shall achieve all this through the design of a small 'System-on-a-Chip' based on the Mentor Graphics *Pyxis* design environment, with a combination of structured practical computer-based exercises, carried out under the supervision of a demonstrator; short 'mini-lectures' to introduce key issues and to provide direction; demonstrations of important techniques and opportunities to measure the actual performance of a real chip.

Format

In this project, you will generally work in pairs. Parts of the project will be done on a shared basis, but each participant will take responsibility for different key parts of the system, later brought together to form the final collaborative design. Each participant will also undertake supporting work away from the terminal (reading, researching, planning and so on); each will submit two interim reports and a final report. These will be written independently, though there will be some common elements.

Schedule

Week 1: Introduction to project objectives and tools. Use of HDL for modelling design concept and system design. Examination and adaptation of transistor-level schematics for a simple 2 input logic gate. Exercises on design of 2 input logic gate. **First interim report**.

Week 2: Development of ring oscillator and programmable divider design using synthesis tools. Creating symbol and schematic diagram for ring oscillator element. Introduction to simulation of schematic design. Estimating system performance. Identification of design errors. Comparing results with measured performance of CMOS ring oscillator circuit. **Second interim report**.

Week 3: Layout and design rule verification of NOR gate structure using *Pyxis Layout*. Layout vs Schematic verification. Estimating parasitics. Netlist and parasitic extraction. Modelling NOR gate performance. Comparing results with measured performance of CMOS ring oscillator circuit.

Week 4: Further detailed modelling using Eldo. Investigating techniques for semi-custom design. Floor-planning, placement of logic blocks, pads and primitive cells, automatic routing and optimisation. **Final report**.

Mini-lectures

(i) Introduction to VLSI Design & HDL. Synthesis. Ring oscillators.
(ii) Design verification by simulation. Integrated Circuit Layout. Design rule checks. Semi-custom layout.

Project Organisation

Some important dates and deadlines are as follows:

Friday 8th May 2015 (Week 3)	Project begins (LR6)
Thursday 14th May 2015 (Week 4)	Hand in date for first interim report
Thursday 21st May 2015 (Week 5)	Hand in date for second interim report
4 pm Friday 5th June 2015 (Week 7)	Hand in date for final project reports
	No reports will be accepted after this date.

During the project period, approximately 8 hours per week will be time-tabled for each project as sessions when one or more demonstrators will be available, to give introductory talks, guidance and help. You are expected to attend all sessions. On the directions of the Teaching Office, a record will be kept by the demonstrators in charge.

You will need to spend some additional 12 hours per week per project working on your own (including report writing). For computer-based projects, some of this time will need to be spent working at spare computer terminals, subject to availability.

You will be issued with a Laboratory Notebook. This is to be used to record *all* day-to-day activities, as a sketch book for any draft design work, to record calculations, results, etc. Demonstrators may ask to see notebooks when marking reports to check that books are used correctly with entries properly laid out and dated.

This project requires 3 reports to be submitted, i.e. 2 short interim reports and a final report. The maximum total length taken together (typed or hand-written A4 pages) must not exceed 14 sides, plus calculations, and drawings.

Interim reports should be posted in the box used for Part II experiments on the landing outside the E & IE Teaching Lab (accessible whenever the Department is open). Final Reports must be handed in at the CUED Entrance Lobby.

In preparing reports, you are expected to adhere to the page limits, and to *keep the volume of appendices to a minimum*. The format will be as follows:

Interim Reports: 2 sides each, excluding appendices.
Final Report: Not greater than 10 sides of A4, excluding appendices.
Further details are provided elsewhere in this sheet and in the document *Third Year Project Guide*.

References

Microelectronics Circuits, 5th Edition , AS Sedra & KC Smith, O.U.P., 2004.

Principles of CMOS VLSI Design, (Second Edition), N Weste & K Eshraghian, Addison Wesley.

Both books are available in CUED Library and in many Colleges.

Introduction to IC Design

Present day semiconductor technology allows designers to build integrated circuits with millions of transistors or logic gates. This has come about largely as a result of the steady refinement of the processes used to manufacture memories, microprocessors and peripheral interfaces, which has made sub-micron device dimensions an accepted fact. This progressive evolution in microelectronics was predicted by Gordon Moore in the sixties, and appears to be continuing. It was realised early in the evolution of microelectronics that design was a limiting factor, and that the unaided human designer simply cannot cope with the complexity of even a few thousand devices, let alone microprocessors or other circuits comprising millions of transistors. Future developments in technology promise to increase this number progressively.

System-on-a-Chip (SoC) is a revolutionary new approach in microelectronics which aims to integrate all the components of an entire electronic system into a single integrated circuit. The resultant circuit or 'chip' may contain digital, analog, mixed-signal or radio-frequency functions, and even micro-mechanical systems – all on one chip. This approach is cost-effective, since it enhances compactness and marketability, and it may also increase the yield or efficiency of the manufacturing process compared with a design involving a circuit board with several chips. However, developments such as these call for new methodologies and tools to address the more challenging design, verification and test problems presented by SoCs in this rapidly evolving area.

Moore's Law describes a doubling of IC complexity every eighteen months, and design productivity has lagged significantly behind this explosive growth. At the same time, competitive forces in the market place are shortening product lifetimes and compelling manufacturers to enhance productivity while coping with an increasing number of products and telescoping the design cycle into a fraction of the time. In these circumstances, designers are increasingly unable to take advantage of available technology in time to meet inexorable market demands. In order to cope with the burgeoning complexity, designers have adopted a number of strategies, several of which we shall meet during this project.

Design Automation
First of all, designers make heavy use of EDA (electronic design automation) - the use of computer-based workstations to store, display and manipulate electronic design data. Elaborate suites of software have emerged to support the activity of IC design, and any designer must acquire proficiency with these in order to be productive. This is a key theme in the VLSI Design project.

Hierarchical Design
Secondly, virtually all designs make use of *hierarchy*. A popular approach - *top-down* design - involves decomposing the design into a *top-level* block and defining the *sub-blocks* required to build it. Each sub-block is then progressively decomposed until the design has been reduced to the level of the most primitive *leaf cells* available. At each stage of the hierarchy, the complexity is maintained at a level that can be handled efficiently. By splitting the design task in this way into smaller, more manageable items which can be systematically designed, the overall design process becomes less daunting. We shall make full use of the hierarchical design approach in this project.

Design reuse
Each block created using the hierarchical approach can be regarded as a resource that can potentially be incorporated in future designs. The *re-use* of such intellectual property (IP) is a key feature of the modern approach to IC design and is regarded as the only way to take advantage of the multi-million device capability of the silicon process. Its implementation across the industry is making formidable demands both of designers and of EDA tool developers. Several of the elements we shall work with in the VLSI project are examples of *re-usable IP*.

Project Activities

The project will begin with an introduction to the role of VLSI design, followed by a brief explanation of the operation and significance of the target System-on-a-Chip, forming the design to be undertaken. This is a *programmable digital divider*, a key element in a digital *frequency synthesiser*, which you will find in any mobile phone, walkie-talkie or other modern communication device. The key parts of this are covered in more detail below.

To summarise, a frequency synthesiser produces an output at one of a selection or pre-programmed frequencies, for use in a receiver or transmitter, to determine the frequency of operation. An oscillator is needed to make this work, and we will use a design known as a *ring oscillator* to provide a regular system clock. There will be a short discussion of how we go about making digital ICs from MOS transistors (covered in detail in module 3B2). The logic inverter will be used as an elementary example to introduce the concepts of schematic representation, net-lists and parasitic components, and to develop the basic concepts of transistor layout, in which the precise geometrical shape and size of the devices are specified.

In order to investigate and predict the behaviour of the target design, we shall explore the use of a hardware description language (HDL) for abstract modelling. This will make use of the Mentor *ModelSim* simulator. This approach is intentionally *abstract*, i.e. it is quite independent of the means by which the design is implemented. While this kind of modelling on its own cannot guarantee that a particular implementation will meet all the demands made of it, it gives a means of better understanding the operation of the design in purely functional terms. Most modern IC designs now commence with a preliminary study based on a hardware description language.

Design Synthesis
These activities will introduce the use of the workstation for describing and synthesising electronic systems in a hierarchical and symbolic way using *Pyxis Schematic* and *Leonardo* packages. The use of symbols to represent standard library parts (e.g. logic gates), or to denote sub-circuits, power supplies, inputs, outputs etc will be illustrated. The importance of *properties* as a means of conveying information about these entities to other 'down-stream' design tools will be stressed.

These tools will be used to develop a circuit representation of the target design, the key parts of a *frequency synthesiser,* which we can regard as a System-On-A-Chip. It comprises four key modules:

- A ring oscillator module. This element provides an oscillatory signal which can be used as a clock waveform to control the timing of other circuit elements. Its

operation will be studied and its design investigated at progressively greater depth as the project proceeds.

- A programmable divider. This element comprises a number of sub-circuits, listed below, and its purpose is to receive an input signal at a high frequency and to generate an output at a frequency which is at a specified sub-multiple of the input. A much fuller description of the rôle of programmable dividers in frequency synthesis is given below, where it is also explained that a complete synthesiser requires some additional components that we won't have time to design.
- The divider module. This element performs a division using bistables connected in cascade to count the input pulses supplied. To make it programmable requires that we can reset it to zero periodically. The counter specified for this design will require 8 or more bits
- A comparator is used to compare the outputs from the counter with fixed inputs to determine when the reset occurs. This is implemented using combinational gates which may include a 2-input XOR.

For convenience, the divider, comparator and other key components are derived from a library of standard parts; however, these are themselves made up of lower level parts (for example, bistables and combinational gates) and their construction can be explored and even altered if necessary.

In addition, the design will make use of pad cells, which provide the physical means in a real integrated circuit for introducing input and output signals as well as power supplies. The design of these is beyond the scope of this project, and standard library cells will be used as supplied.

Note that we shall take largely for granted the method of operation of the digital circuits used, though, as mentioned below, it is important to verify that they are correctly translated into valid physical representations.

The facilities in *Pyxis Schematic* for checking the electrical correctness of the circuit will then be explored. Although it cannot be established at this stage that the circuit will work in the way anticipated by the designer, it is possible to identify elementary mistakes, such as missing connections, short-circuits, and so on. *Pyxis Schematic* will also be used to examine the detailed, transistor-level structure of the key element within the ring oscillator, a 2-input NOR gate.

Simulation and Design Verification

Because IC fabrication is a costly and time-consuming activity, it is vital that a design be verified as functionally correct before the fabrication process is begun. Numerical modelling plays a vital role in this, through the use of *simulators*, which attempt to describe the characteristics of an electronic system in terms of numerical models. This procedure is nowadays an accepted phase of any non-trivial electronic design project (even if it does not involve the design of ICs), and a wide range of simulation tools exist. In some, the level of modelling employed is incredibly detailed; as a result, only moderately complex circuits can be investigated without unacceptable penalty in terms of computing time. With others, the models used are simpler, but give less accurate results. They can be used to check for correct operation in much larger designs. In practice, a number of different approaches to simulation may be required, even within the compass of a single design. In this project we shall use two

different simulation packages: *ModelSim* and *Eldo* – members of the Mentor *Questa ADMS* mixed-signal simulation package.

A well-established approach to verifying the correct operation of a design involves capturing a *schematic* representation of the design, which can then be modelled using a selection of simulators, to confirm that it responds, electrically, as well as in other functional ways, as the designer intended. In most approaches to design, as here, the schematic is created at an early stage as it allows conceptual problems to be resolved before any effort is wasted on layout.

The *Eldo* simulator may be used to model in a very accurate way the detailed behaviour of a single 2-input NOR gate (of which there are many within the ring oscillator module), providing detailed parametric information, transient response and other performance metrics that could not be obtained from *ModelSim*. However, *Eldo* could not realistically be applied in this way to problems involving more than perhaps a few hundred transistors, and could never be used to verify the correctness, in every subtle aspect, of a typical entire design. However, by a suitable partitioning of the simulation task into digital and analogue sections, mixed signal simulators such as *QuestaADMS* can apply detailed analogue models where they are essential, while relying on simpler digital models for elements whose performance is well understood.

A combination of the *ModelSim* and *Eldo* simulators will be used to check for correct operation of the entire design, and to give an estimate of its anticipated performance, in terms of oscillation frequency and the various output waveforms to be expected.

Circuit Layout

The detailed design of the ring oscillator continues with the specification in terms of *mask layers* for the 2-input NOR gate. A *mask specification* comprises the set of plans by which a manufacturer creates the physical representation of an integrated circuit, and determines in the most minute way the performance of the system. We shall gain familiarity with the layout tools available in the Mentor *Pyxis Layout* application, by completing the design of a partly-constructed gate - a 2-input NOR gate - which will serve also to introduce the key mask layers and the tools available for modifying them. Techniques for interconnecting complex structures using metal and polysilicon layers will be explored. The importance of *design rules* as a fundamental constraint in the design process will be illustrated, using directed exercises to introduce the tools available (*InstantDRC* and *Calibre*) for detecting design rule violations.

The schematic can be compared, almost literally wire by wire, and device by device, with the corresponding layout, in a procedure known as LVS (Layout versus Schematic), in order to confirm that the layout is actually an accurate representation of the schematic. This uses the *Calibre* module of the *Pyxis Layout* package.

Methods for functional verification of the circuit will also be explored, using available tools (*Calibre*) for extracting netlists and parametric values (for example, transistor dimensions, parasitic capacitances, etc) from layout.

Semi Custom Design

After successfully completing the introductory exercises, you will design a gated ring oscillator, using a chain of 2-input NOR gates connected as logic inverters.

A compact layout, free from design rule violations and with the highest possible oscillation frequency will be the primary objectives. The final phase of the project involves the use of the semi-custom design tools available with the Mentor Graphics suite to prepare the design to the standard required for fabrication (although we shall not be able to fabricate the designs for this project). This will include floor-planning, automated cell placement and automatic routing, and will result in a layout containing input and output pads, power supply pads and power frame, and all wiring. A full-colour check plot will be generated for incorporation in the final report.

Measurement of performance

Although as already explained it will not be practicable to fabricate the designs created during this project, it is still important to explore how well the performance of manufactured devices matches that predicted by the numerical simulation techniques used to verify the designs. This stage would be equally as important were the designs to be manufactured. We shall achieve this by comparing the measured performance of a ring oscillator circuit designed in a previous project and fabricated commercially. A special breadboard will facilitate the provision of power supplies and test waveforms, and we shall use standard laboratory equipment (voltmeter, oscilloscope, counter) to carry out the measurements.

Schedule

The schedule of activities outlined on page 1 of this document indicates a natural sequence of operations for this work. There are also a number of paper design exercises that need to be carried out in advance of the corresponding workstation sessions. Apart from this, there is a certain amount of flexibility over the order in which the various activities can be carried out, and it is also possible for groups to vary the amount of time spent on certain parts of the project.

Certain parts of the project make quite heavy demands on computing resources - in particular, *Eldo* and *Calibre* - and it will actually be advantageous for there to be some variety over the class in the activities being pursued at any instant.

In addition, a session of experimental measurement forms a key part of the project, and requires about 2-3 hours. Three sets of equipment will be provided, and all groups are expected to take part. The experiment can be carried out *at any time* during the second, third or fourth week since it does not depend explicitly on the development of the design, and does not require the use of the workstation, and the measured results are required in the final report only.

Part IIA Third Year Projects

Standard Project in VLSI Design SB1

Getting Started

This pamphlet provides information on getting started (logging in, setting up to run Mentor Graphics tools and executing *Pyxis* or other Mentor applications). It also covers issues such as the framework on which the tools are based, the *Acrobat* Browser (on-line manual system), and generating printed hard copy of designs.

Logging in

During scheduled sessions, specific clusters of workstations are reserved for use by the project. These are: **tw6*xx*** and **tw5xx** (parts), both near the centre of the DPO. In addition it should be possible to use **tw9*xx*** (at the east end) where the seating may be more comfortable; but these are not specifically reserved. These are set up to run Linux (Centos distribution) as per the normal departmental standard. Mentor itself requires a Linux Redhat platform, and is therefore hosted on two separate servers, named **clae01** and **clae02**, into which users will login from these terminals. The choice of server is made randomly to spread the load; they are identical.

Special '**mentor*xx***' work spaces have been set up and are dedicated to this project, for use by the pairs of students involved. You access these using **your own** user IDs. They are provided for this work only, and should not be used for any other activity. A CamTools-based Project Worksite (including a bulletin board and other facilities) will be available to allow participants, demonstrators and staff to communicate and exchange information in various ways. You will need to monitor this from time to time, as it will be used to publish information not in the Lab Guides and not easily obtained anywhere else.

Logging in to use the Mentor suite involves three stages:

- Logging in to the workstation
- Secure login to **clae01**
- Setting up the Mentor environment and starting *Pyxis Project Manager*

Logging in to the workstation – in the DPO, this means via one of the allocated clusters. At the **tw-** terminal, login, in the normal way, giving your user ID and password. You will be presented with a standard Gnome terminal session.

Secure login to clae01 – a dedicated start-up command has been installed to streamline this process. See the Quick-Start Leaflet issued on day 1. Otherwise this is accomplished with the following command:

```
ssh -X clae01 [or clae02]
```

This should open up a Bourne-again shell '*bash*' on the **clae01/clae02** server.

Give the command: **`hostname`** if you want to verify this.

As mentioned above, special 'mentor*xx*' work spaces have been set up and are dedicated to this project, for use by the pairs of students involved. You access these using **your own** user IDs. Either member of the team may login.

Please note that your own personal login directories are not shared, and you are expected to maintain normal security measures, such as keeping your personal password private, etc.

Your shared ***mentorxx*** workspace is provided with a number of directories to support this project. Some are empty, but others contain data files that should help you get started. The **cbt** directory is the receptacle for most of the design files needed, and it is the place where your own (and your partner's) files will be held. The **mgc** directory is for special technology files used by the Mentor tools, should they be required. A **bin** directory is provided, and contains a selection of specially written startup scripts, one entitled **pyxis**. You should normally use this to begin your Mentor Graphics session, using the procedure described below.

Before you can start Mentor, you must change to the directory tree where your shared designs are kept. The command required is:

`cd /groups/IIA/ptiivlsi/mentor`*`xx`*

where *xx* is the suffix for the workspace you and your partner have been allocated.

A dedicated startup script has been provided to carry out necessary housekeeping chores at the start of your session. This sets up important system parameters and initialises menus and other features that assist the Mentor Graphics environment. Other scripts depend on these parameters and will fail if the startup utility is not run. The script is found in the **bin** directory in your shared workspace. To invoke this, give the command:

`source bin/pyxis`, followed by **Enter**.

Among other things, the script will set up a variable called **MHOME** which is the path to your personal shared workspace – see above. Once it has run, the command:

`cd $MHOME`

will return you to this directory. We shall use this notation later on to identify directories and other data items you need to access.

The script will carry out a few more housekeeping operations, finally changing directory to the directory in which your design work will be accomplished. Normally this will be **cbt**. It is possible to change directories from within any Mentor Graphics application, but file access problems may be reduced if you adhere to this advice.

As it signs off, this utility will print out a list of commands which you may find helpful later when you need to invoke Mentor tools.

Please bear in mind that you are expected to carry out the project during the scheduled periods in the DPO, which is the base for all project resources. Outside scheduled Mentor project times the allocated clusters are liable to be in use for other projects. It will normally be possible, subject to availability of a suitable terminal, to login from other DPO workstations. The procedure just described should work from any of these. The demonstrators may be able to advise on the procedure for logging in from other terminals, but note that they will only be available during scheduled sessions.

It is also possible (though it may not be a trivial exercise) to access the Mentor software from a remote (non-CUED) terminal - for example, a networked College PC running a suitable Windows-based X client, e.g. NoMachine. Such an arrangement

could offer potential for access at times when the DPO may not be accessible. However, note that you are expected to attend the scheduled sessions in the DPO in real and not virtual form! We will give what help we can to those who wish to try remote access, and we will post what information we gather on this topic on the Moodle project support platform (see below), but there are no guarantees of success.

Using Mentor Design Tools

Designs are essentially collections of Unix files and directories. However, unless you have been advised otherwise by a demonstrator, you should **not** use the standard Unix commands (like **cp**, **mv**) to manipulate design files. If you do, you are likely to encounter problems with **broken** or **unresolved references**, which may mean the loss of your work. Instead, use the facilities provided in the tools themselves for copying or moving designs.

Shell scripts have been provided to automate the setup for starting the Mentor *Pyxis* application. Unless you are advised otherwise, give the command: **dmgr_ic** to run it. When the graphics window is presented, use the mouse to resize it to a convenient format. A separate pamphlet (Lab Session 1) is provided to introduce you to the features available in *Pyxis* Project Manager, and demonstrators will normally be available to assist in the event of queries.

Acrobat Browser (On-line Manual Facility)

The Mentor Graphics System does not include printed manuals. Instead, a comprehensive on-line manual system is provided, based upon a PDF archive which is continuously available to the project. It contains facilities for display of detailed information on a very wide range of Mentor-related topics, many of which are not relevant to this project. In addition it offers powerful search facilities that let you identify all available on-line manuals containing relevant references. You can start the *Acrobat Browser* either directly, or from most of the other tools. A separate pamphlet (Lab Session 1) briefly describes the use of the *Acrobat Browser*, and an introductory laboratory session demonstrates its use. In addition, most of the tools have a Help menu that gives you access to shorter segments of the manuals presented in a web browser.

There will be demonstrations and screenshots of the tools shown during the accompanying mini-lectures.

Generating hardcopy

Most of the Mentor tools provide facilities for printing out schematics, symbols, layouts and simulation results. You will probably wish to include printouts of this kind in your interim and final reports. Note that most of the tools generate output in the form of Postscript files, and special post-processors are provided where required to prepare these for transmission to the printer. In some cases it is possible to preview output on the screen, and decide at that stage whether or not to commit it to paper.

Both monochrome and colour printing are available, and output is made available against individual users' quotas, using the standard departmental procedures. It is up to the individual user to determine the proportion of colour printing undertaken. None is strictly necessary for the project, though users may want to produce a colour plot of the final layout produced, for example, for the Final Report.

Please be sure to monitor the Moodle Project Support Platform for late-breaking news.

The printing facilities available that you may wish to use are described below. You may need to refer to this page during later lab sessions.

Reports/Documents in text files
In an *xterm*, use the following command:

```
lpr –P Uniflow <filename>
```

It is also possible to use the clipboard to email text to yourself/your partner for later incorporation in other documents. See on for details of using CamTools to move data to other computing systems e.g. your own personal computer, using CamTools.

VHDL Source Code or other text files – using gedit
Go to **File > Print**
Select Uniflow as the print destination, or alternatively use Print To File to produce a document in PDF format

Digital waveforms from EZWave

Option 1
Go to **File** > **Print**
Select **Uniflow** as printer
Select **A4** as paper size
*Click **OK***

Option 2
Save waveforms as .jpg file to view/ print later or incorporate in document
Go to **File** > **Export**
Enter a filename
Click **Save**

Schematics in *Pyxis Schematic*
For *Pyxis* Schematic, it is best to save in postscript format first, and to preview any Postscript file before printing.

Go to **File** > **Print**
Enter filename in the **Output Filename** space
Select **Export to File** as **Destination**
Select **A4** as **Select Page Size**
Select **Basic Postscript** as **Output Format**
Click **OK**

Use the Nautilus utility to preview the result by navigating to the output file, right click, and choose the Document Viewer tool. When ready to print,

Go to **File->Print**
Select Uniflow as the print destination, or alternatively use Print To File to produce a document in PDF format

Layouts in *IC Studio*
For *Pyxis* Schematic, it is best to save in postscript format first, and to preview any Postscript file before printing.

With the layout visible on the screen,
Go to **File** > **Print** to invoke the **Print with Pyxis Plot** dialogue.
Click the **Export to File** radio button to set this option
Enter Uniflow into the **Þrinter Name** box
Click on **Formatter Options**, and choose the ***Raster Postscript*** option; ensure the **Colour Output** checkbox is ticked if you require colour. Enter an output file name –

for example, your design's name – into the **Output File**: box (use the browser to locate a suitable directory if you wish); enter **ps** into the **File extension** box.
Ignore the **plotcap file** box.
OK the dialogue box.
Next, OK the **Print with Pyxis Plot** dialogue box.

Use the Nautilus utility to preview the result by navigating to the output file, right click, and choose the Document Viewer tool. When ready to print,

Go to **File->Print**
Select Uniflow as the print destination, or alternatively use Print To File to produce a document in PDF format

Where necessary, more detailed instructions about preparing to print from Mentor applications are given in later **Lab Guides**.

Note: If you have any difficulties arising from printing and the need to generate hardcopy output, please consult one of the demonstrators; they may refer you to the Computer Operators. We will publish any new information about printing options on the Project Worksite.

Transferring files to other systems

You may wish to transfer files – for example .png, .tiff, .jpg, .txt or .ps files – to other networked computer systems for compiling reports etc. There are several ways to achieve this – here is just one, based on use of CamTools. An alternative flow based on Moodle is also being developed.

First, open a Teaching System web browser and navigate to CamTools; login.
Select **Personal Tools**, then **Upload Files**
Click on **Select Files to Upload**
Navigate to `/groups/IIA/ptiivlsi/mentorxx`
Select the file or files;
Click **Start Upload**

On your own PC/Mac etc, open a web browser and navigate to CamTools; login.
Select **My Files**
Open, download, save the files as required.

Exiting from Mentor Applications and logging out

When you finish using any Mentor application, you should exit from it, in order to avoid its presence slowing down the system for others and for yourself as you run other applications. It is especially important to close down the Mentor Graphics system in an orderly way when you logout, for similar reasons. Of course, it is possible to minimise a tool while you temporarily work with a different one, then return to the original application. You may find this style of operation very useful.

Most of the Mentor tools now provide a convenient way of exiting, via the **MGC > Exit** command. (There are one or two exceptions). The recommended way to leave a tool is to close down all its design child windows (first saving any important data) by double-clicking the Select (left) mouse button on their system icon (top left). To quit the tool completely, use the **Exit** command provided in the **MGC** menu.
Important note: You should close down all other Mentor applications before you close down *Mentor Pyxis* Project Manager, and finally close down your terminal session.

Warning: If you do not exit from tools in the approved way, you are liable to leave behind a collection of unwanted processes which may slow down the system. If you think this has happened, consult a demonstrator for advice on how to proceed.

Once you have exited from all Mentor and other applications and closed all shell windows, you will be left with a blank login screen. Close this, and finally log out from the terminal session.

Project Support Platform and Electronic Mail

A dedicated Project Support Platform hosted on the Moodle VLE has been developed to support this project. This is accessible at any time and from any workstation or networked PC with a suitable browser. It is open to all participants, demonstrators and staff, and offers a convenient way of sharing information among participants. It offers a Wiki, available to all, so that if you wish to document a particular feature not otherwise covered, you are welcome. There is a discussion board and other interactive facilities, as well as a handy calendar and schedule. We should appreciate feedback of the effectiveness of these tools, as well as any suggestions to enhance their usefulness.

Information will also be posted here in the form of advice on the following topics:

- First Interim Report
- Second Interim Report
- Experiment on Electrical Testing of the Ring Oscillator
- Accessing Mentor from outside CUED
- Hints and Tips

The information provided is expected to grow as the project progresses.

Please remember you are welcome to access the Moodle Support Platform from your own account at any time and from anywhere you have web access.

Entry Point URL: `https://www.vle.cam.ac.uk/`
Title: `Part IIA Project: SB1: VLSI Design`

While engaged in the project, you should get into the habit of checking the Moodle support platform on a regular basis, since this mode of communication will be used for speedy distribution of information e.g. reminders about deadlines, minor amendments to the lab sheets, etc. We recommend you start up the *Mozilla Firefox* browser during your session so you can monitor Moodle continuously. This is most important outside scheduled periods, when the demonstrators will not be available.

If you run into problems outside the scheduled sessions, you should post details of the difficulty to the Moodle VLSI Design Forum. This is monitored by all the demonstrators on a regular basis, and while we cannot guarantee turn-around on solutions for problems reported in this way, we will do our best. If you think your posting on the Forum has not been seen or acted on, you may send a **short** message to all demonstrators to draw its existence to their attention. For this purpose **only**, the special email address **mentor-staff@eng.cam.ac.uk** has been provided. **Please do not send long technical messages or copy-paste text from your Mentor session to this email address** – the demonstrators have been briefed to ask you to resubmit to the Forum, to allow them to deal with it**. Please do not send emails to demonstrators' personal addresses**.

Standard Project in VLSI Design SB1

Design of logic gates in CMOS

This pamphlet gives a simple introduction to the principles of operation of CMOS logic gates, and attempts to point out some of the performance trade-offs that have to be addressed in their design. It also introduces the ring oscillator, which forms the heart of one of the design elements of this project.

The CMOS inverter

The CMOS inverter or NOT gate is the simplest of the many boolean function that can be implemented using MOS transistors, but it serves to establish the fundamental principles which are common to all types. Our aim is to represent the logical values **1** and **0** by some electrical quantity and devise circuits that will allow us to manipulate these in the same way that boolean functions transform the logical values. It is conventional to represent logic **1** with a high voltage, and logic **0** with a low voltage. Over some forty years of evolution of transistorised logic, the industry has settled on standard values of 5 volts and 0 volts for this purpose. But there is nothing particularly magical about these values, and indeed, within the last few years there has been a progressive shift to lower voltages: 3.3 V, 3 V, 1.8 V …

In CMOS we use *enhancement mode* MOS (metal-oxide-silicon) FETs. These devices have no conductive channel (i.e. they are OFF) until the gate-source potential exceeds a specific value, the threshold voltage V_T. When the channel is fully formed, the device is said to be ON. The table below summarises the dependence of these devices on the gate-source voltage V_{gs}. The threshold voltages given are typical.

Type	V_T	$V_{gs} = 0$	$V_{gs} = 5$	$V_{gs} = -5$
n-type	+1v	OFF	ON	OFF
p-type	–1v	OFF	OFF	ON

The availability of *complementary* devices - that is, p-type and n-type - is one of the distinctive features of this technology. It is important to remember that the current equations for the p-type device involve inequalities of the opposite sign to those for the n-type device. In the n-type device, conduction occurs when p-type majority carriers are *repelled* from the vicinity of the channel, requiring that the gate be positive with respect to the source in order to switch the device ON. In the conductive p-type device, n-type carriers are repelled, requiring that the gate be negative relative to the source to turn the device ON. Thus a p-type device is turned OFF by a high gate voltage which turns an n-type device ON, and the converse holds for a low gate voltage. This complementary operation is of course the key to CMOS circuit design.

A simple model

The static characteristics of the CMOS inverter have been covered in Part IA and in third-year courses, and their derivation will not be repeated here. The theory of the devices themselves, and the approaches to fabrication required to make them work as fast as possible have already been covered in a Part IB Elective course, which many of you will have attended. In this project we shall use industry-standard simulation tools to model the behaviour of the devices and the circuits in which they operate. However, in order to be able to have an understanding of the results we observe, we shall develop a very simple approximate model for the conductance G of the MOS transistor channel, from which we can deduce many aspects of logic gate performance.

Consider the simple rectangular MOS transistor channel of length and width L and W respectively, shown in Fig. 8 below. Suppose the mobility of carriers within the material is μ.

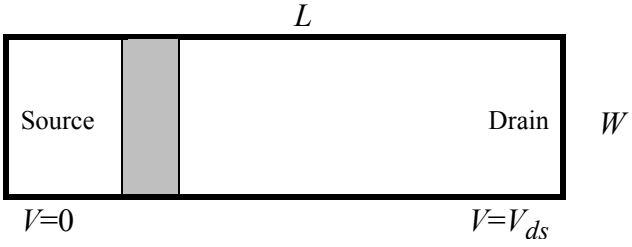


Figure 8 – Model MOS transistor channel

Assume that the gate voltage $V_g = V_{dd}$, so that the device is conductive.

Let the charge density in the shaded element near the source be Q per unit length.

Then $Q = C_{ox}\, V\, W$

where V is the excess voltage on the gate, and is given by $V = V_{dd} - V_T$.

If V_T is much smaller than V_{dd}, we can approximate this:

$$Q = C_{ox}\, V_{dd}\, W$$

The current in the channel, I is given by:

$$I = \mu\, Q\, E$$

If we may assume that E is invariant along the device, we can write:

$$I \approx \mu\, Q\, V_{ds} / L \;=\; \mu\, C_{ox}\, V_{ds}\, V_{dd}\, W/L$$

Hence the conductance $G = I/V_{ds}$ is given by:

$$G = \mu C_{ox}\, V_{dd}\, W/L\ .$$

The result just obtained can be interpreted as follows. The channel conductance G (with the gate potential above threshold) is proportional to the factor μC_{ox} , often referred to as the *process gain factor*, and denoted by K. This is determined by material and manufacturing considerations, and is not under the direct control of the circuit designer. G is also proportional to the supply voltage V_{dd} , within the limits of the approximation, and - most importantly from the perspective of the designer - to a purely geometric factor W/L, often known as the *aspect ratio* of the MOS transistor.

Layout and performance of logic gates

We shall now use the simple result deduced above to predict some of the characteristics of the logic inverter, and shall then explore how this treatment can be extended to more complicated gates.

The circuit schematic and layout of a CMOS inverter are shown in Figure 9 below.

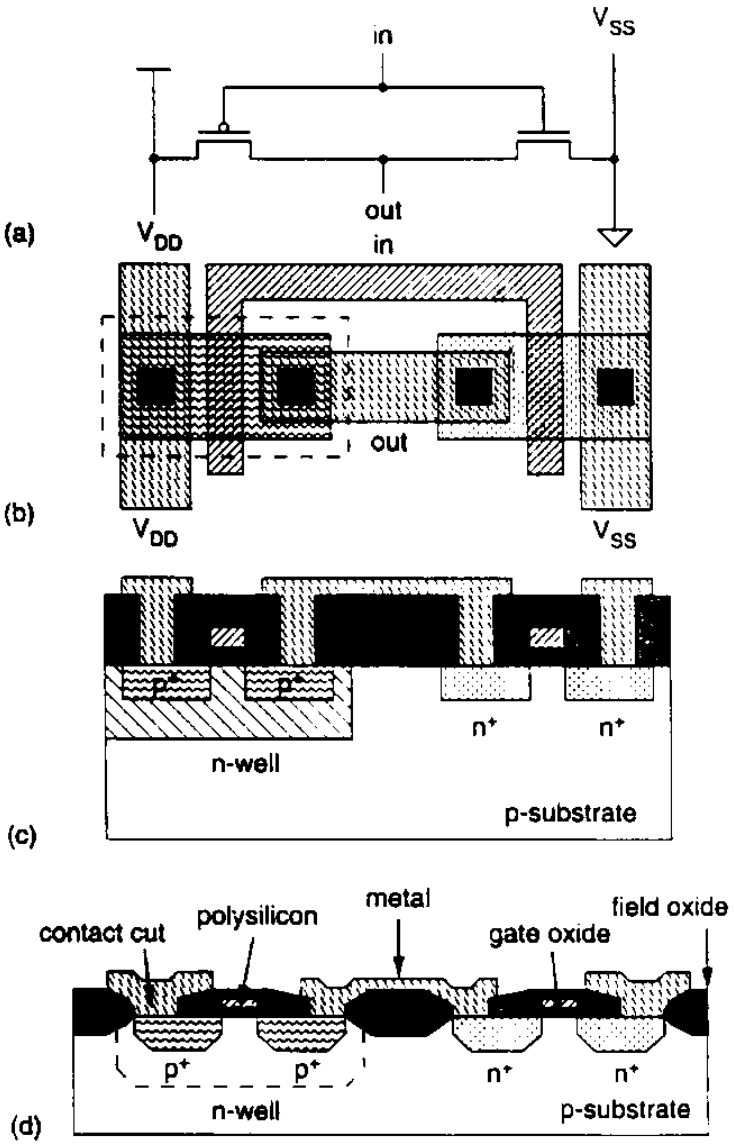


Figure 9 – The CMOS inverter

The transistor channel is formed by the superposition of the polysilicon gate electrode over the thin gate oxide which is grown on the surface of the lightly doped semiconductor itself. The source and drain are formed from highly doped p- or n-type material created using ion implantation.

It can be seen that the channel width *W* is determined by the width of the thin oxide region (that is, the distance it extends into the plane of the paper), while the channel length *L* is actually defined by the *width* of the stripe of polysilicon that extends across the channel. Both these quantities are directly under the control of the designer, and are in fact the primary means by which the designer determines the performance of each element of the system.

A logic circuit may be characterised by three key parameters:

- Physical size
- Power consumption
- Speed of operation

An ideal circuit will have minimum size, fastest speed and the lowest power demand. It turns out, however, that it is not possible to optimise all characteristics simultaneously, and the designer must weigh his requirements characteristics and balance the competing constraints to suit the application.

Physical size

This is proportional to the product of *W* and *L*. Minimising the values of *W* and *L* clearly give the smallest size. However, this approach is limited by the ability of the manufacturing process to produce reliable devices at the smallest dimensions. Like any manufacturing process, it is subject to tolerances which set a lower limit to the size that can be achieved. For example, the current minimum channel length presently available from commercial fabrication facilities is about 0.05 – 0.7 μm (though many designs use greater channel lengths). The designer is thus constrained by a set of rules (*Design Rules*) that govern the dimensions of the various structures he specifies.

Speed

When the input to the inverter switches between 0 and 1 or *vice versa*, the output terminal is expected to change state correspondingly. The metal and semiconductor elements connected to the output all possess capacitance to the substrate (and to all other conductors, to be pedantic). We shall regard these separate contributions as a single *lumped* capacitance *C* connected between the output and ground. This capacitance must be charged and discharged through the conductive channel of one or other transistor: the discharge to 0V via the n-type channel, and the charge-up to *Vdd* via the p-type channel. Although there are other factors that determine the delay experienced by a signal in travelling between the input of a logic gate and its output, under most circumstances it is the processes just outlined that dominate. It is possible to develop accurate equations relating the drain current to gate and drain potentials. These can be integrated to determine how long the charge or discharge processes take. However, in the interests of simplicity we shall regard the system as a set of switched conductances which control the charge/discharge dynamics, and we shall assume that the conductances are constant, and defined by the simple conductance model given earlier.

Using classical RC network theory, we can now write down expressions for the delay for *rising* and *falling* edges at the output terminal. We may also assume that the gate potential changes abruptly, so that the channels also change state abruptly between conductive and non-conductive. A convenient pair of expressions, which are well borne out by experiment, are as follows:

Delay time for rising edge $\approx 3\ C / G_p$ *Delay time for falling edge* $\approx 3\ C / G_n$

As expected, the smaller the value of C, the shorter the delays and the faster the circuit. Considering the expressions for G, it is clear that the higher the supply voltage V_{dd}, the more rapidly it will switch.

From the IC designer's point of view, the larger the value of W, the higher the conductance and hence also the faster the circuit. There is a subtlety here which we must not overlook. The capacitance C is to some extent dependent on the values of L and W, since the gate itself and the depletion regions forming the drains of the transistors may make a sizable contribution to the total parasitic capacitance at the output terminal. Increasing W may not necessarily produce the speedup that the simple expression suggests.

There is yet a further complication. Although the values of V_{dd} and C_{ox} are usually the same for both n- and p-type transistors, the mobility μ differs significantly between the types. Typically, the ratio μ_n / μ_p may be two or more. This means that for given values of W and L, the conductances of n- and p-channel devices will be different by the same factor. An inverter using identically-sized transistors would therefore have different delays for rising and falling edges. In real logic circuits, asymmetric delays of this kind are a disadvantage, and measures must be taken to equalise them. For the inverter this is easily achieved by adjusting the ratio W/L for each device such that $\mu W/L$ is constant. To keep the devices compact, we may use the minimum permissible value of L, and scale W. This results in the value of W for the p-channel being greater than that for the n-channel device by a factor μ_n / μ_p .

Power

Power is the rate of consumption of energy. One of the great virtues of the CMOS inverter, is that provided the input terminal is held either at logic **0** or logic **1**, only one transistor is conductive, the other being non-conductive. As a result there is never under these circumstances a direct pathway for current to flow from V_{dd} to ground. This means in its turn that the power consumed is negligible. These considerations have led to the use the use of CMOS in many applications where lower power operation is essential.

This convenient model for power dissipation in CMOS is in fact too simple. Charge does actually flow from V_{dd} to ground every time the output terminal switches from logic **1** to logic **0**. While the output terminal is high, the parasitic capacitance charges up to a potential V_{dd}, and holds a charge $C\ V_{dd}$. When the output terminal changes state, this charge is conducted to ground through the n-channel transistor. A fixed packet of charge $C\ V_{dd}$ is thus transferred from power supply to ground each time the input of the gate changes state from **1** to **0** and back again. If the gate is driven by a periodic waveform of frequency f, we can identify a flow of current from V_{dd} to ground, via the parasitic capacitance, whose average value I_{av} is given by:

$$I_{av} = f C V_{dd}$$

Hence the average power consumption by this single gate is VI_{av}, or $f C V^2$. We can apply this kind of reasoning to entire CMOS integrated circuits, provided we know the parasitic capacitances associated with each gate output. Estimating the frequency of operation of every individual gate may be rather more difficult. As a first approximation it may be satisfactory to assume that each gate is driven at some fraction (say, about one half) of the externally-supplied clock signal connected to the circuit, on the basis that gates will be clocked at frequencies ranging from DC right up to the clock frequency, and in roughly equal quantities. Where more information is available about signal frequencies, a more accurate dissipation figure can be determined.

This form of *dynamic* loss is the principal mechanism by which dissipation of power occurs in CMOS. At sufficiently high frequencies, the amount of heat produced can be enough to require special measures to disperse it (the Pentium and DEC Alpha chip are well known illustrations, and dissipate several watts because they run at several GHz). This is a further reason to try and minimise the parasitic capacitances. Note that the power dissipation is also proportional to V^2. Hence increasing V_{dd} in an attempt to improve switching speed has the disadvantage of significantly increasing the dynamic power dissipation. Many new, high-speed devices are designed to run at power supply voltages of 1 V or even less for this very reason.

More complicated gates

The arguments just developed for the CMOS inverter can be extended to other more complicated combinational gates. Any arbitrary combinational gate in CMOS can be expressed as a *pull-up* network (consisting of p-channel MOS transistors, and connected to V_{dd}) and a *pull-down* network (consisting of n-channel transistors, and connected to ground), as in the diagram below.

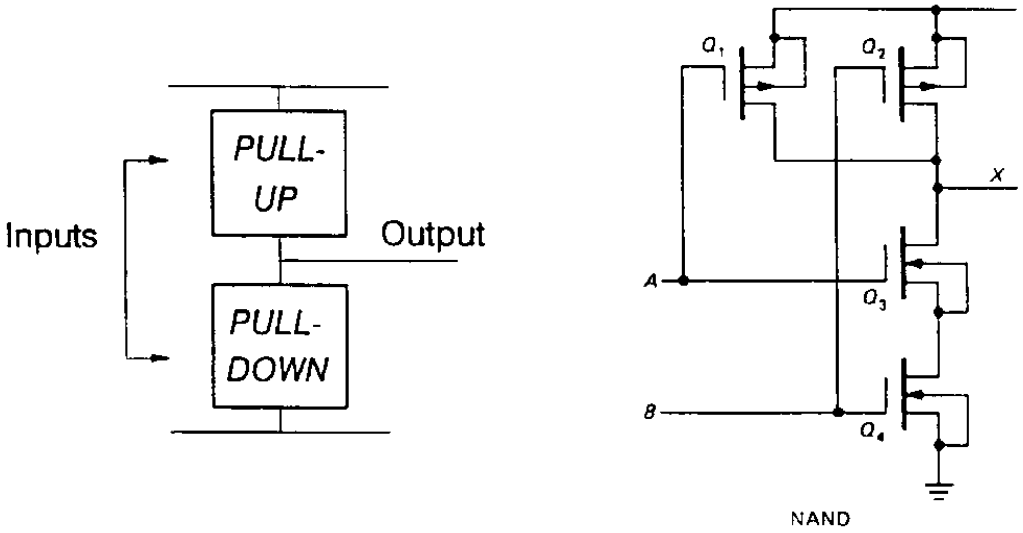


Figure 10 – More complex gates

In CMOS each input signal controls a device in the pull-up network and another in the pull-down network. The networks are designed in such a way that **either** the pull-up network connects the output to V_{dd}, **or** the pull-down connects the output terminal to ground. The networks must never be simultaneously conductive.

For example, the transistor schematic for a two-input NAND gate is also shown in the diagram, comprising two n-channel transistors in series (pull-down) and two p-channel transistors in parallel (pull-up).

Let us consider now the issue of delay in more complicated gates. Firstly, the existence of additional MOS transistors connected to the output terminal increases the parasitic capacitance observed there. This is an inescapable fact. Secondly, we see that our simple model for charge/discharge of this capacitance through a single p-type or n-type device has to be extended. In the NAND gate, the discharge takes place through two n-type devices Q3 and Q4 in series. Of course, *both* devices must be conductive to allow this to take place. However, the charging current may flow through Q1, Q2, or Q1 **and** Q2, since these devices are parallel-connected. We need to reconsider how to determine the overall conductance of the pull-up and pull-down networks when these may consist of arbitrarily complicated groupings of transistors in series/parallel. A reasonable way of calculating effective conductances for series-connected devices is to use Ohm's Law. Since the 'resistance' of a transistor is $1/G$ and is therefore proportional to L/W, we can argue that for a string of n series-connected transistors:

$$\left[\frac{L}{W}\right]_{effective} = \left[\frac{L}{W}\right]_1 + \quad \ldots \quad + \left[\frac{L}{W}\right]_n$$

This is actually not such a good approximation as one might expect, because it fails to take account of the observed non-linear nature of channel conductance, which is dependent on the source and drain voltages as well as on the gate potential. In other words, the conductance of each of two (physically identical) transistors with identical gate voltages, connected in series, may *not* be identical, since they are quite likely to have different source and drain voltages. This is a fairly fundamental limitation of our simple conductance model. However, we shall ignore this problem for now, as to do otherwise would tend to obscure the important fundamental principles we wish to emphasise.

With parallel-connected transistors, we have observed that the conduction may take place through one or more devices, according to the state of the inputs. When estimating the delay times for a gate, it is normal to use worst-case assumptions. These may then be used to determine worst-case figures for the delay through an entire system, a conservative approach to design. Hence the effective conductance for a parallel set of devices may reasonably be taken as the conductance of just one of those devices, representing the worst case.

Applying this rationale to the 2-input NAND gate, we see that the worst-case pull-up conductance is given by:

$$\mu_p C_{ox} V_{dd} \left[W_p / L_p\right]$$

with only one p-channel device conducting, while the pull-down conductance is:

$$0.5\,\mu_n C_{ox} V_{dd} \left[W_n / L_n\right]$$

since *two* n-channel devices are in series. In order to equalise the *worst-case* rising and falling delays, we must set these equal. It follows from this that:

$$\left[W_p / L_p\right] = 0.5 \frac{\mu_n}{\mu_p} \left[W_n / L_n\right]$$

A further issue which we shall mention but not develop fully, is how we might attempt to normalise the rising and falling delays of gates with multiple inputs to those of the inverter. There may be worthwhile advantages in a family of logic gates with roughly uniform delay, and it is apparent that we can at least improve the situation by choosing $\left[W_n / L_n\right]$ for multi-input gates so that the worst-case conductance in the pull-down network is similar in magnitude to that for the inverter. This in its turn affects the aspect ratio chosen for the p-channel devices.

We should note that selecting the transistor dimensions in this way cannot in general match rising and falling delay times for *every* possible input transition. For the 2-input NAND, the delay for a rising input will be a factor of two faster than the worst case if both inputs are switched simultaneously from logic **1** to logic **0** rather than just one.

The Ring Oscillator

The ring oscillator is at least conceptually one of the simplest forms of oscillating circuit. It has a number of technical limitations that render it less than ideal for general-purpose oscillatory circuits, but it has some fascinating features which are of great value to the integrated circuit design engineer.

An ideal ring oscillator consists of a homogeneous chain of identical inverting gates connected in cascade - that is, with the output of each stage connected to the input of the next. The output of the final gate is connected back to the input of the first, giving a closed ring-like structure. See the diagram below.

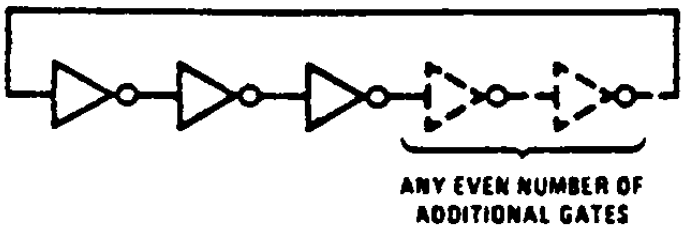


Figure 12 – Ring oscillator: the simple implementation

Note that the circuit shows an *odd* total number of gates. The simplest way to understand how such an oscillator works is to imagine that any convenient point - say, the input to gate number 1 - attains the value logic **0** at time 0. Since each gate inverts, the output from gate 1 will become logic **1** a short time after (dependent on the delay imposed on the signal as it passes through gate 1. The output from gate 2 will become logic **0** a further gate delay later, and so on. It is possible to visualise a transition (logic **1** to logic **0**) racing around the ring, being inverted and delayed as it progresses. When the transition reaches the output from the *n*th gate (that is, the input to gate 1), it has undergone an *odd* number of inversions, and therefore is of the opposite polarity, i.e. logic **0** to logic **1**, and arrives at time t_1 determined by the delay through *n* gates. The inverted transition races around the ring and reaches the starting point after a further delay t_2, and is now of the same polarity as the initial transition. It is clear that so long as the signals assume regular logic levels, the circuit has no stable state, and will continue to oscillate.

To determine the oscillation period, it is necessary to know the delay through each stage. From above we see that during a complete cycle of oscillation a transition must travel round the ring two complete times. If all stages have identical delay τ both for rising and falling edges, it follows that the period of oscillation is $2n\tau$. Hence the frequency of oscillation is:

$$f = 1 / 2n\tau$$

Since there is such a close link between τ and the frequency of oscillation, the ring oscillator has considerable value in the design of integrated circuits. It provides a convenient way in which the delay of a simple gate can be measured to a high degree of accuracy with comparatively simple instrumentation. This information is vital in order to provide a check on the fabrication process, and also to determine proper numerical models for the devices. The highest speed gates currently available have delays of a few tens of picoseconds. To measure this kind of delay directly is extremely difficult. Moreover, even if instrumention were available with a suitable response, the capacitive load it would impose on the gate under test would grossly distort the measurement.

The solution is to build a ring oscillator comprising a suitably large number of identical gates. With the ring oscillating, the period of oscillation may be measured and the expression above used to determine the individual gate delay. Any combinational gate capable of performing an inversion may be used in place of the inverter shown. Hence this technique can be applied to NAND, NOR or even XOR and XNOR gates to measure their performance. Virtually all integrated circuits nowadays have a small area set aside for a *parametric 'drop-in'* - manufacturers' jargon for a small piece of circuitry - typically including a ring oscillator - inserted specifically to allow verification that the processing has complied with expectations.

Examples - solutions should be included in your First Interim Report

1. Referring to the section above: **Design of Logic Gates in CMOS**, show how to draw the pull-up and pull-down networks in such a way as to produce CMOS gates which perform following logic functions

   (a) $Y = \overline{A.(B + C + D)}$

   (b) $Y = \overline{A + (B.C.D)}$

2. How must the transistors be dimensioned to achieve equal *worst-case* delays for rising and falling edges with the gates (a) and (b) considered above. Assume that μ_n / μ_p is 2.5, and that *minimum* allowed values for L and W are 2μm and 4μm respectively.

3. Comment on the ring oscillator method for determining the delay through a gate. What, if any, are the advantages and disadvantages of this approach to delay measurement compared with a more direct technique?

4. Determine the frequency of oscillation of a ring oscillator if each of the gates used has delay τr for rising edges and τf for falling edges.

5. How might you vary the frequency of a ring oscillator? Hint: consider the simple expressions for delay and transistor conductance developed earlier in this sheet.

6. Is it possible for a ring oscillator to oscillate at any other basic frequency than that given by the expression: $f = 1 / 2n\tau$? Explain your reasoning.

7. Should it be possible for a ring oscillator with an even number of gates oscillate? Under what limited conditions might oscillation be detectable?

A practical ring oscillator circuit

The simple ring oscillator circuit outlined above well illustrates the principles of operation, but it is not a particularly practical arrangement. For convenience, we shall wish to have some way of switching on or off the train of oscillations, and we shall require more than one output so we can observe the phase relationships between them. To achieve this we require at least one gate to have a second input (i.e. NAND or NOR) so that an external signal can be applied, as in the improved circuit below.

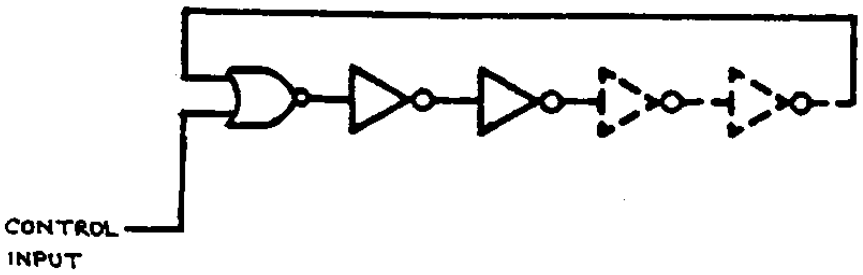


Figure 13 – Ring oscillator: a more practical design

To maintain homogeneity in the ring, and for convenience, we shall use 2-input NOR gates for every stage of the ring; where only a single input is required, we shall simply wire together the two inputs A and B, giving the functionality of an inverter. A library part – nor2x - will be used in the first instance to provide us with a feel for the way the circuit behaves, but in the final design we shall create a mask layout to implement our own version of the 2 input NOR gate, and investigate how its performance compares with that of AMS's design.

The required specification for the ring oscillator part of the design is provided in the following section.

Standard Project in VLSI Design SB1

Design Specification

As outlined in an earlier section, the target design is a complete digital system-on-a-chip for a frequency synthesiser. A fuller explanation of frequency synthesis methods is given later, and there may be opportunities for some designers to incorporate more elaborate features, provided the basic operating principles have been understood. The basic design contains the following elements:

- A 2-input NOR gate (schematic already provided) for introductory experiments.
- A ring oscillator module, which provides a master clock.
- A counter module. This element comprises a binary counter which counts clock pulses input via the clock terminal. The schematic design shown below illustrates a six-bit counter, but this is purely indicative – your design will be driven by the specification developed later in this section
- A comparator module, which compares the outputs of the counter with a pre-programmed number, applied by means of a set of input pins on the chip, and generates an output when these match.
- Additional logic to respond to the comparator's output and reset the counter; a further element is included so the output waveform is symmetric.

There is considerable scope for individual creativity within the above specification, but each element is dealt with in more detail below. You can read about the architecture of frequency synthesisers later in this chapter.

The first stage of development involves creating a representation of the entire system using a hardware description language (HDL) to explore and verify its operation. Figure 1 presents a block diagram showing the organisation of the main elements.

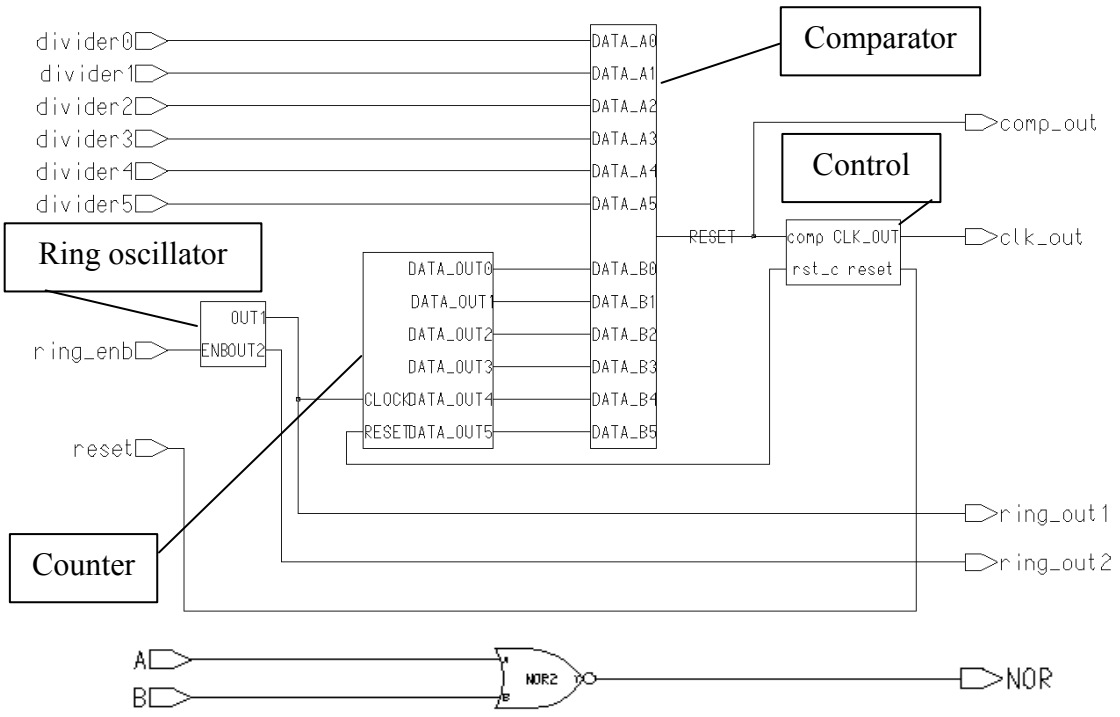


Figure 1 Block Diagram of complete **top_level** system-on-a-chip

Ring Oscillator

A key element of this project is the ring oscillator, **ring_oscillator**, whose development is intended to highlight all the various activities involved in integrated circuit design. During the project you will construct a definition in a hardware description language for it. You will develop a schematic representation of its constituent gates and a graphical symbol; you will have a chance to predict its performance using digital and analogue simulation techniques. You will design and verify the mask layout for the 2 input NOR gate used in its construction, and finally, you will use the **ring_oscillator** module as a signal source in a programmable divider design which includes counters and other sequential logic devices.

The following section describes the specification to which we shall work for the **ring_oscillator** part of the design.

Ring Oscillator Specification

The simple theory of the ring oscillator is given in the *Design of Logic Gates in CMOS* pamphlet, and the relation between gate delay and oscillation frequency is derived.

The *initial* design for **ring_oscillator** will use the NOR2 part whose detailed specification is provided by AMS; the salient details are given overleaf. We shall aim to produce a ring oscillator with a single Enable input **ENB**, and two outputs **OUT1** and **OUT2** taken direct from the **ring_oscillator** module, with a stated oscillation frequency and phase difference between them. In order to specify the basic characteristics of the ring oscillator portion of the design, we shall use the criteria detailed opposite.

A single instance of the NOR2 gate is also shown in the block diagram in Figure 1. This gate is entirely separate from the ring oscillator and all other parts of the design. It is included to provide a straightforward means of introducing some of the major concepts related to schematic design and simulation.

Later on we shall develop our own NOR2 gate at the most detailed level possible, using individual MOS transistors. The result will be a gate which is broadly similar to the AMS part, but whose delay characteristics will likely be quite different. With a little reflection and research, and with not too much effort you should be able to design a part which works **significantly faster** than the original library part, and you should be able to identify ways of making the design more compact (and hence cheaper).

Ring Oscillator Specification

Approximate target operating frequency 80 MHz
based on AMS NOR2 part with V_{dd} = 3.3 volts and nominal loading capacitances
(**Note**: you will refine this ball-park figure as you develop your design).

Input ENB = **1** Oscillator disabled
Input ENB = **0** Oscillator enabled

Typical input/output waveforms

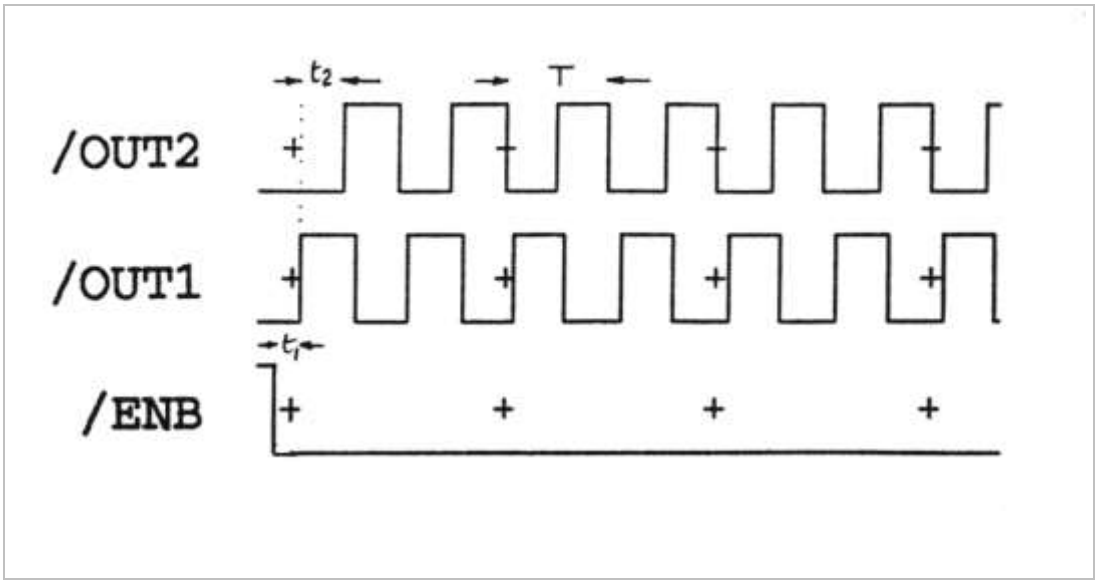


Figure 2 Input and output waveforms

Detailed timing specifications (see waveforms)

Parameter	Description	Lower limit (ns)	Nominal value (ns)	Upper limit (ns)
T	**Oscillation Period**	**11.0**	**12.5**	**14.0**
t_1	**Delay from ENB to OUT1**	**1.0**	**2.0**	**1.0**
t_2	**Delay from OUT1 to OUT2**	**3.0**	**4.0**	**3.0**

Note that the Oscillation Period *T* corresponds to the *target operating frequency* given at the head of this page.

NANGATE

NOR20

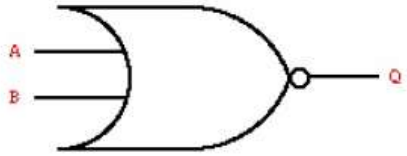
www.ams.com

Databook Build Date: Wednesday Jun 13 12:43 2012

Copyright © 2004-2010 Nangate Inc.

Conditions for characterization library **c35_CORELIB_TYP**, corner **c35_CORELIB_TYP_TYPICAL**: Vdd= **3.30**V, Tj= **25.0** deg. C .
Output transition is defined from **10%** to **90%** (rising) and from **90%** to **10%** (falling) output voltage.
Propagation delay is measured from **50%** (input rise) or **50%** (input fall) to **50%** (output rise) or **50%** (output fall).

Strength	20
Cell Area	54.600 um^2
Equation	Q = "(B+A)'"
Type	Combinational
Input	A, B
Output	Q



State Table		
A	B	Q
L	L	H
H	-	L
-	H	L

Propagation Delay [ns]					
Input Transition [ns]		0.05		1.50	
Load Capacitance [fF]		1.00	80.00	1.00	80.00
A to Q	fall	0.08	0.60	0.16	0.80
	rise	0.11	0.93	0.20	1.06
B to Q	fall	0.04	0.57	0.09	0.77
	rise	0.07	0.91	0.25	1.14

Output Transition [ns]					
Input Transition [ns]		0.05		1.50	
Load Capacitance [fF]		1.00	80.00	1.00	80.00
A to Q	fall	0.17	1.25	0.42	1.35
	rise	0.22	2.19	0.43	2.23
B to Q	fall	0.09	1.20	0.33	1.31
	rise	0.22	2.19	0.38	2.22

Capacitance [fF]	
A	6.0000
B	6.0000

Leakage [pW]
0.25

Input Pin Power Consumption [nW/MHz]			
Input Transition [ns]		0.05	1.50
A	fall	28.00	27.00
	rise	-24.00	-25.00
B	fall	14.00	12.00
	rise	-7.00	-6.00

Dynamic Power Consumption [nW/MHz]					
Input Transition [ns]		0.05		1.50	
Load Capacitance [fF]		1.00	80.00	1.00	80.00
A to Q	fall	56.00	50.00	104.00	66.00
	rise	57.00	61.00	109.00	72.00
B to Q	fall	17.00	16.00	67.00	34.00
	rise	44.00	57.00	106.00	68.00

Figure 3 – AMS C35 NOR20 gate characteristics

Digital sub-systems

In addition to the ring oscillator core, we shall design and explore various additional digital sub-systems. The theme chosen for this year's project is a frequency synthesiser, a key building block in communications systems and other applications. Design of an entire frequency synthesiser is beyond the scope of a short project like this, unfortunately, so we shall confine ourselves to developing the digital parts only.

As for the ring oscillator, we shall initially investigate and verify the design using HDL. We shall then consider its implementation. Unlike the ring oscillator, which we shall design and model at the most detailed level possible (creating it from individual MOS transistors), the frequency synthesiser design will wherever possible take advantage of pre-defined library components. These will include:

- A digital counter constructed from components taken from the AMS library (combinational gates and D-type bistables). The counter's Clock input may be driven by one of the ring oscillator outputs, and the outputs are used to supply clock waveforms needed in other parts of the design. This will emphasise the value of the hierarchical approach to design.

- A programmable divider, which will be developed to fulfil the requirements for the frequency synthesiser case study being targeted in this project. There is a degree of individual choice available in the specification of this part. The programmable divider can itself be divided into two parts: a counter, and a comparator to allow detection of when the counter enters a pre-programmed state. Additional counters will also be needed to meet the specification.

- The ring oscillator already described fulfils the role of a variable voltage-controlled oscillator (VCO). Other elements not being designed here include:

    - a phase-sensitive detector (PSD) In most frequency synthesisers, this is a critically designed analogue multiplier whose detailed design alone would take more time than available for the entire project! It is possible to implement a PSD using an XOR gate, however, and groups making good progress and wishing to push the limits are free to experiment with this idea.

    - a master oscillator. Such a device is normally implemented using an off-chip circuit including a high-stability quartz crystal, typically operating at 1MHz or some other convenient frequency. Where necessary, we will be able to use simulated signals for this purpose.

    - a low-pass filter – another critical element normally calling for off-chip components, and an elaborate and time-consuming design procedure.

Frequency synthesisers

A frequency synthesiser is an electronic system for generating any of a range of frequencies from a single fixed-frequency timebase or oscillator. They are found in many modern communications devices, including radio receivers, mobile telephones, Bluetooth accessories, radiotelephones, walkie-talkies, CB radios, scanning receivers, satellite receivers, GPS systems, musical instruments, etc. A typical application is illustrated in Fig. 4 below, where it is seen that a frequency synthesiser is used as a Local Oscillator in a two-way radio system, providing channel selection by digital control. However, the list of possible applications extends far beyond the short list of examples given.

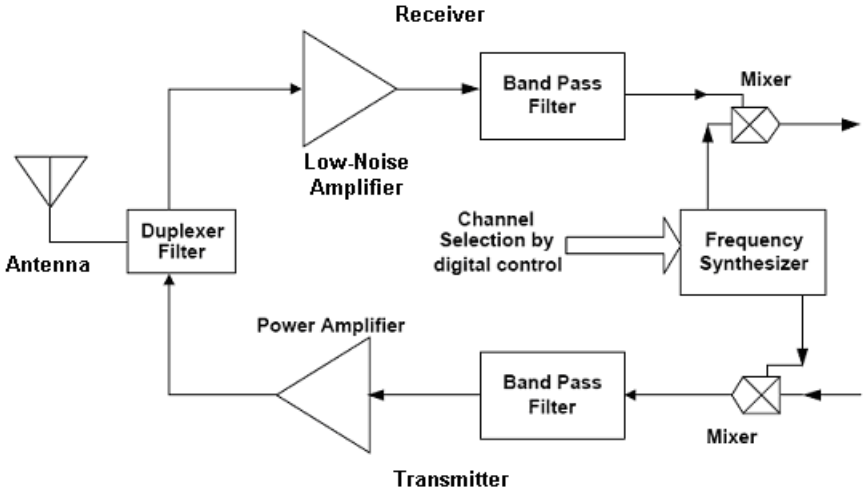


Figure 4 A communications application for a frequency synthesiser

Evolution

Prior to widespread use of synthesisers, radio and television receivers relied on manual tuning of a local oscillator, usually by means of a variable capacitor. The availability of *varactor diodes,* in which a reverse biased p-n junction exhibits a capacitance dependent upon the applied bias, made it possible to miniaturise such systems. However, variations in temperature and aging of components caused frequency drift. Automatic frequency control (AFC) solved some of the drift problem, but manual retuning was still often necessary. Since transmitter frequencies are well known and very stable, an accurate means of generating fixed, stable frequencies in the receiver was desirable to solve the problem.

A simple and effective solutions employs the use of many stable resonators or oscillators for each tuning frequency. Quartz crystals offer good stability and have often been used for this purpose. However, this approach is practical when only a handful of frequencies are required. It quickly becomes costly and impractical in applications where many frequency channels are required. For example, the FM radio band in many countries supports 100 individual frequencies from about 88 MHz to 108 MHz. Cable television can support even more frequencies or channels over a much wider band. A large number of crystals increases cost and requires more space.

Many solutions to these requirements have been devised over the years. Some approaches include phase locked loops, double mix, triple mix, harmonic, double mix divide, and direct digital synthesis (DDS). The choice of approach depends on a

number of factors, including cost, complexity, frequency-step size, switching rate, phase noise, and permissible spurious output levels.

One set of techniques, known as *coherent techniques*, generate frequencies derived from a single, stable master oscillator. In most applications, the use of a crystal oscillator is widespread, but other forms of resonator and frequency source can be used. In another approach, *incoherent techniques*, it is possible to obtain different frequencies from a set of several stable oscillators, typically through frequency multiplication, division, and summing/differencing (mixing). The vast majority of synthesizers in commercial applications use coherent techniques because they offer much greater flexibility and can readily be implemented as integrated circuits to give a compact, low-cost solution.

Synthesisers used in commercial radio receivers are almost invariably based on phase-locked loops or PLLs. Many types of frequency synthesiser are available as integrated circuits, reducing cost and size. High end receivers and electronic test equipment use more sophisticated techniques still, often in combination.

Principle of PLL synthesisers

The block diagram in Fig. 5 below shows the basic elements and arrangement of a PLL-based frequency synthesiser.

A *phase-locked loop* does for frequency what the *Automatic Gain Control* does for stabilisation of voltage or power output in a radio receiver. It compares the frequencies of two signals and produces an error signal which is proportional to the difference between the input frequencies. The error signal is used to drive a voltage-controlled oscillator (VCO) which generates the required output frequency. The output signal is fed through a frequency divider where it is divided down in a known ratio, and fed as one input (the *comparison frequency*) to a *phase-comparator*, which compares the phase of the divided signal with the fixed comparison frequency from the master oscillator. The phase-comparator output, in effect an *error signal*, is passed through a carefully designed low-pass filter, and fed back to the input of the system to control the operating frequency of the VCO. This arrangement constitutes a negative feedback loop. If the output frequency drifts, the error signal will increase, driving the frequency in the opposite direction so as to reduce the error.

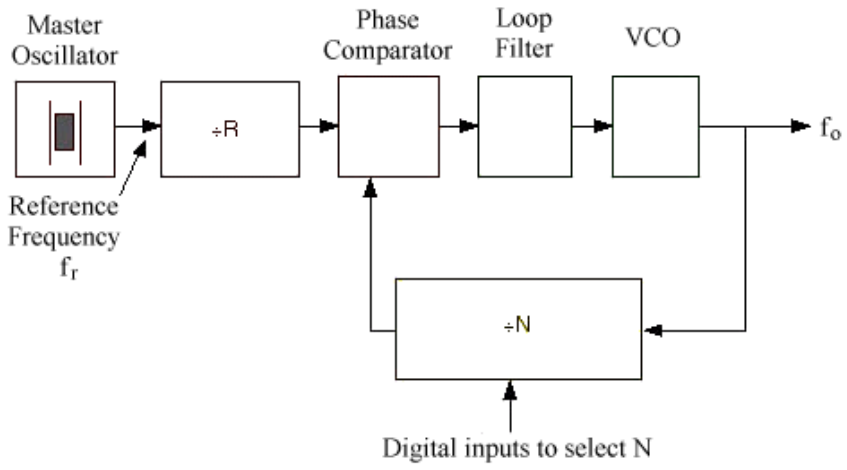


Figure 5 Classic phase-locked-loop (PLL) frequency synthesiser

Thus the output becomes *locked* to the frequency at the other input. This input is derived from a crystal oscillator or similar, which is very stable in frequency, and referred to as the *reference frequency*. Phase-locked loops have many applications both in communications and other branches of electronics; for example:

- Frequency synthesisers for digitally-tuned radio receivers and transmitters
- Demodulation of both FM and AM signals
- Recovery of small signals that otherwise would be lost in noise (lock-in amplifier)
- Recovery of clock timing information from a data stream such as from a disk drive
- Clock multipliers in microprocessors that allow internal processor elements to run faster than external connections, while maintaining precise timing relationships
- DTMF decoders, modems, and other tone decoders, for remote control and telecommunications

The key to the ability of a frequency synthesiser to generate multiple frequencies is the divider (÷*N*) placed between the output and the phase comparator, as shown in the simplified sketch below. This usually takes the form of a digital counter, with the VCO output signal acting as a clock. The counter is preset to some initial count value, and counts down at each cycle of the clock signal. When it reaches zero, the counter output changes state and the count value is reloaded. Figure 5 below illustrates the waveforms expected in a simplified design with the single divider *N* set to divide by 4.

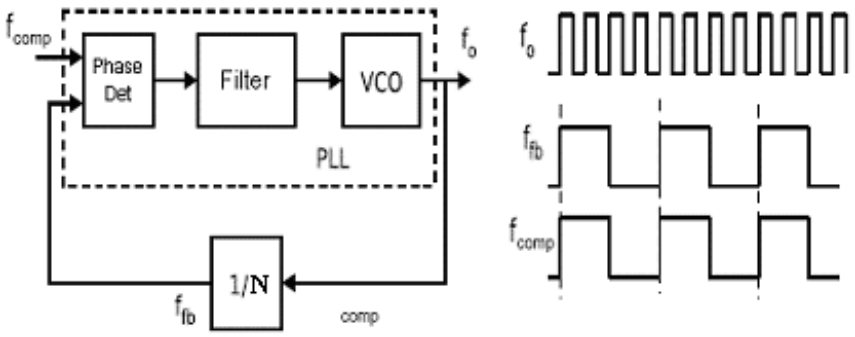


Figure 5 Waveforms with N set to ÷4

This circuit is straightforward to implement using bistable devices (e.g. J-K or D-type flip-flops), and because it is digital in nature, is very easy to interface to other digital components or a microprocessor. This allows the frequency generated by the synthesiser to be easily controlled by a digital system.

Example

Suppose the reference signal is fixed at 1 MHz, and the fixed divider *R* is set to divide by 10; its output frequency is thus 100 kHz. Assume also that the divider *N* can be preset to any value between 1 and 100. The error signal produced by the comparator will only be zero when the output of the divider is also 100 kHz. For this to be the case, the VCO must run at a frequency which is 100 kHz × the divider count value, *N,* or $f_r \times N/R$. Thus it will produce an output of 100 kHz for a count of 1, 200 kHz for a count of 2, 1 MHz for a count of 10 and so on. Note that only whole multiples of the comparison frequency (which is the divided reference frequency) can be obtained.

Practical considerations

In practice this type of frequency synthesiser cannot operate over a very wide range of frequencies, because practical comparators have a relatively limited bandwidth and

may also suffer from aliasing problems. This would lead to false locking situations, or an inability to lock at all. Furthermore, it is hard to make a high frequency VCO that operates over a very wide range. However, in most systems where a synthesiser is used, we do not seek a huge range, but rather a finite number of channels over some defined range, such as a number of radio channels in a specific band.

Many radio applications require frequencies higher than can be directly input to the digital counter N. To overcome this, the entire counter could be contructed using very high-speed logic such as Emitter-Coupled Logic (ECL), or more commonly, using a fast initial division stage called a *prescaler* which reduces the frequency to a manageable level. Since the prescaler is part of the overall division ratio, a fixed prescaler can cause problems designing a system with narrow channel spacings which are often essential in radio communication or broadcast applications. This difficulty can be overcome using a *dual-modulus prescaler*, discussed below.

Further practical aspects concern the amount of time taken for the system to switch from channel to channel, the time to achieve lock when first switched on, and how much noise – random fluctuations in the amplitude, frequency or phase - there is in the output. All of these are a function of the *loop filter* of the system, which is a low-pass filter placed between the output of the frequency comparator and the input of the VCO. Typically, the output of a frequency comparator is in the form of short error pulses, but the input to the VCO must be a smooth, noise-free DC voltage. Any noise on this signal naturally causes unwanted frequency modulation of the VCO. Heavy filtering will reduce this effect, but will lead to a VCO which is unacceptably slow to respond to changes, causing drift and slow response time ; on the other hand, insufficient filtering will produce noise and other problems with harmonics. Thus the design of the filter is crucial to the performance of the system and is, in fact, the main challenge facing the designer of such a system.

A more versatile approach to frequency division

The use of a *prescaler* in a frequency synthesiser to alleviate some of the problems of frequency selection has already been mentioned. A *Prescaler* is an electronic divider circuit used in high-frequency synthesiser designs to overcome the problem of generating signals at frequencies too high to be passed directly through the feedback loop of the system.

A basic frequency synthesiser as described above generates an output frequency f_o, given by the reference frequency f_r multiplied by the division ratio *N/R* such that :

$$f_o = f_r \times N/R$$

Since *N* is an integer, the output frequency is necessarily restricted to whole multiples of the divided reference frequency f_r/R. Typically, these will be the channels for which the radio equipment is designed for, so f_r and *R* will usually be equal chosen so that f_r/R is equal to the channel spacing. For example, for narrow-band radiotelephones, a channel spacing of 12.5 kHz is typical. This is achieved by an appropriate choice of the master oscillator frequency, and the divider *R*, for example 1MHz and 80, respectively.

Suppose that the divider *N* is only able to operate at a maximum clock frequency of 10 MHz, but the output f_o is required to be in the range of hundreds of MHz range.

If we were to interpose a fixed prescaler with a value, *M*, of say, 40, we may now lower the output frequency easily into the operating range of the divider *N*. However, this has introduced an additional factor of 40 into the equation, so the output frequency is now:

$$f_o = 40 \times N \times f_r / R$$

If the comparison frequency f_r / R were to remain at 12.5 kHz, we would obtain channels at a spacing of 40 × 12.5 kHz or 500 kHz, in effect only every 40th channel. However, if we were to reduce f_r by a further factor of 40 to compensate, using an additional divider, the comparison frequency would become 312.5 Hz, which is much too low to allow satisfactory filtering and lock performance characteristics. It would also mean greater complexity in the programming of the divider, as only those ratios giving true channels must be selected, not those increments of 1/40th of a channel that become available with the extra divider stage

An enhanced frequency synthesiser

It is relatively straightforward to incorporate a prescaler in an enhanced form of frequency synthesiser. A *dual-modulus* counter is one which is designed such that it can divide by one of two different factors, usually *M* and *M*+1 – for example, 2 and 3; 8 and 9, etc. The division factor actually applied is determined by means of a control input.

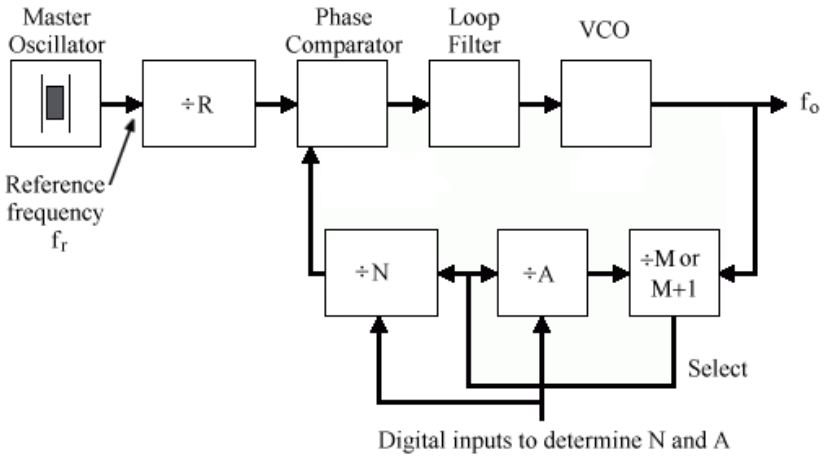


Figure 6 Frequency synthesiser with integer dual-modulus prescaler

In the enhanced design, shown in Fig. 6, the main divider is split into two parts, the primary part *N*, and an additional divider *A*, which has a lower count range than *N*. Both parts of the main divider are clocked from the output of the dual-modulus prescaler, but only the output of the *N* divider is fed back to the comparator. This arrangement is sometimes known as a *pulse-swallow* architecture.

Initially, the prescaler is set to divide by $M + 1$. Both *N* and *A* count down until *A* reaches zero, at which point the prescaler is switched to give a division ratio of *M*. At this point, the divider *N* has completed *A* counts. Counting continues until *N* reaches zero, which represents a further $N - A$ counts. At this point the cycle repeats. Thus:

$$f_o = f_r(M(N - A) + A(M + 1))/R$$

which reduces to

$$f_o = f_r\,(MN + A)/R$$

Hence, while the output frequency f_o is still related to f_r by a factor which depends on *MN*/*R*, a further term *A* is now added. Only the dual-modulus prescaler needs to be constructed from high-speed parts; and the output frequency channel spacing remains the same as the comparison frequency f_r/R.

The diagram below shows the elements and arrangement of a frequency synthesiser with dual-modulus prescaler.

A and *N* can be computed from the following formulae, which relates them to the required division ratio *V*:

$$f_o = f_r/RV$$
$$A = V \bmod M$$
$$N = V \operatorname{div} M$$
$$V = (MN + A)$$

For this arrangement to work satisfactorily, *A* must be strictly less than *M*, as well as being less than or equal to *N*. These restrictions on the value of A imply that not every imaginable division ratio *V* is in fact possible unless the counter settings are carefully chosen. For example, *V* may not fall below $M\,(M-1)$; and with some choices for M, N and A, certain channels will be unavailable.

It is possible to take this idea still further and develop from it a design that is capable of achieving outputs at frequencies that are *fractional* multiples of the comparison frequency. This may be achieved by passing the VCO signal through the ÷(N+1) counter for a part of the counting period, and through the ÷N counter for the remainder. In this way an *average* division ratio of somewhere between N and N+1 is obtained, and hence there is a fractional relationship between the input and output frequencies. This can be achieved with only relatively minor changes to the integer pulse-swallow architecture described, and represents a significant improvement to the basic design, as it means that frequency synthesisers can be used to generate accurately specified, closely-spaced channels at very high operating frequencies in the GHz region. This is beyond the scope of the project, however.

A demanding real-life application

Modern communication protocols tend to allocate closely located channels at very high frequencies. This poses difficult challenges for the system designer. For example, the Bluetooth short-range wireless protocol allocates 79 channels between 2.401 GHz to 2.479 GHz resulting in a channel spacing of 1MHz. The phase noise (and other distortions from the local oscillator used) must be low enough not to cause interference with transmissions on adjacent channels. However, a stand-alone oscillator tunable over a 79 MHz band would not have sufficiently high stability. Moreover, quartz crystals, regarded as essential in applications where frequency must be accurately determined, do not have resonance frequencies as high as 2.4 GHz. The frequency synthesiser architecture allows these challenges to be met, and integrated solutions are now readily available with chip costs of only a pound or so.

Notes on other circuit elements

The Phase Comparator

As seen, a key element in a frequency synthesizer phase-locked loop is the phase comparator. This compares the phase of the divided VCO to that of the divided reference signal. There are several types of phase detector.

The simplest is an exclusive OR gate, which maintains a 90° phase difference, but it is limited in its effectiveness unless the inputs are already at nearly the same frequency. A more complex approach uses a simple state machine to determine which of the two signals has a zero-crossing earlier or more often. This brings the PLL into lock even when it is off frequency. This type is known as a phase-frequency detector.

An analogue four-quadrant multiplier, also known as a mixer, can be used as a phase detector. Multiplying the VCO and comparison signals generates an output consisting of a low-frequency signal whose amplitude is related to the phase difference, or phase error, between the VCO and the comparison, plus a second (unwanted) signal at twice the oscillator frequency that can be eliminated by means of a low-pass filter.

The Voltage-controlled Oscillator (VCO)

The VCO must be capable of generating a signal of variable frequency, governed by the value of an external control voltage. A common way to achieve this is by use of an LC oscillator, comprising an LC "tank" circuit, which oscillates by charging and discharging a capacitor through an inductor. Most LC oscillators use off-chip inductors, since on-chip inductors suffer large resistive losses, and are of low Q. A voltage-controlled capacitor is one method of making an LC oscillator vary its frequency in response to a control voltage. Any reverse-biased semiconductor diode displays a measure of voltage-dependent capacitance and can be used to change the frequency of an oscillator by varying a control voltage applied to the diode. Special-purpose variable capacitance varactor diodes are available with well-characterised wide-ranging values of capacitance.

A ring oscillator can alternatively be used as a VCO. The frequency is controlled by varying either the supply voltage or the capacitive loading on each stage. Such VCOs generally have poorer effective Q than a well-designed LC oscillator, and so suffer more jitter and instability than other types. However, the ring oscillator approach offers the advantage of requiring no off-chip components or on-chip inductors. They may also have larger tuning ranges than other kinds.

The loop filter

Key characteristics of a practical frequency synthesizer include: the time taken for the system to switch from channel to channel, time to lock when first switched on, and how much noise appears at the output. All of these are a function of the loop filter. This is a low-pass filter placed between the output of the frequency comparator and the input of the VCO. In the most basic systems, a simple RC network may be adequate. Typically the output from the frequency comparator is in the form of short error pulses, but the input of the VCO must be a smooth noise-free DC voltage. Any noise on this signal will cause unwanted frequency modulation of the VCO. Heavy filtering will make the VCO slow to respond to changes, causing drift and slow response time, but light filtering will produce noise and other problems with harmonics. Hence the design of the loop filter is critical to the performance of the system and its optimisation may consume a great deal of design time.

Design Choices

During the course of this project you will need to make a number of design choices at different stages. A detailed specification has been given for the performance of the ring oscillator design, and one of the themes of this project is the interplay between the details of layout and the performance that can be achieved. This will be investigated in a later Laboratory Session, and in the laboratory experiment organised to accompany this project.

However, we want to leave you with a greater degree of choice than this in the development of the digital part of the design. From the foregoing it is clear that there must be several viable approaches to the design of a programmable divider suitable for use in a synthesiser. Rather than specify these in minute detail, we will present an outline specification for the required operating frequency, and allow you to investigate the options and make the necessary design choices, referring as necessary to the notes above. You are of course welcome to discuss these options with a Demonstrator; you should be able to establish their feasibility at an early stage using functional simulation (Lab Guide 2), and it will be possible to adapt them (after discussion with a demonstrator). For example, you might conclude that a different Master Oscillator frequency will suit the needs of your project better than the one proposed. If your design might need additional inputs or outputs to/from the chip, you should also discuss this with a Demonstrator

You are expected to include in your First Interim Report as detailed a specification as possible of the target design your group has converged upon, and should include a block diagram clarifying details of those elements you will design, and those which will be provided externally.

Frequency Synthesiser Specification

Design Parameter	**Specification**
Number of discrete channel frequencies	85
Frequency band of operation *	26.965 – 27.405 MHz
Channel frequency spacing	10 kHz
Master Oscillator reference frequency	1 MHz
Dual modulus counter (if required)	÷2, ÷3

* This range of frequencies corresponds to the Eu specification for the Citizens' Band allocation. All generated channels are to be within this frequency range to avoid interference to other services.

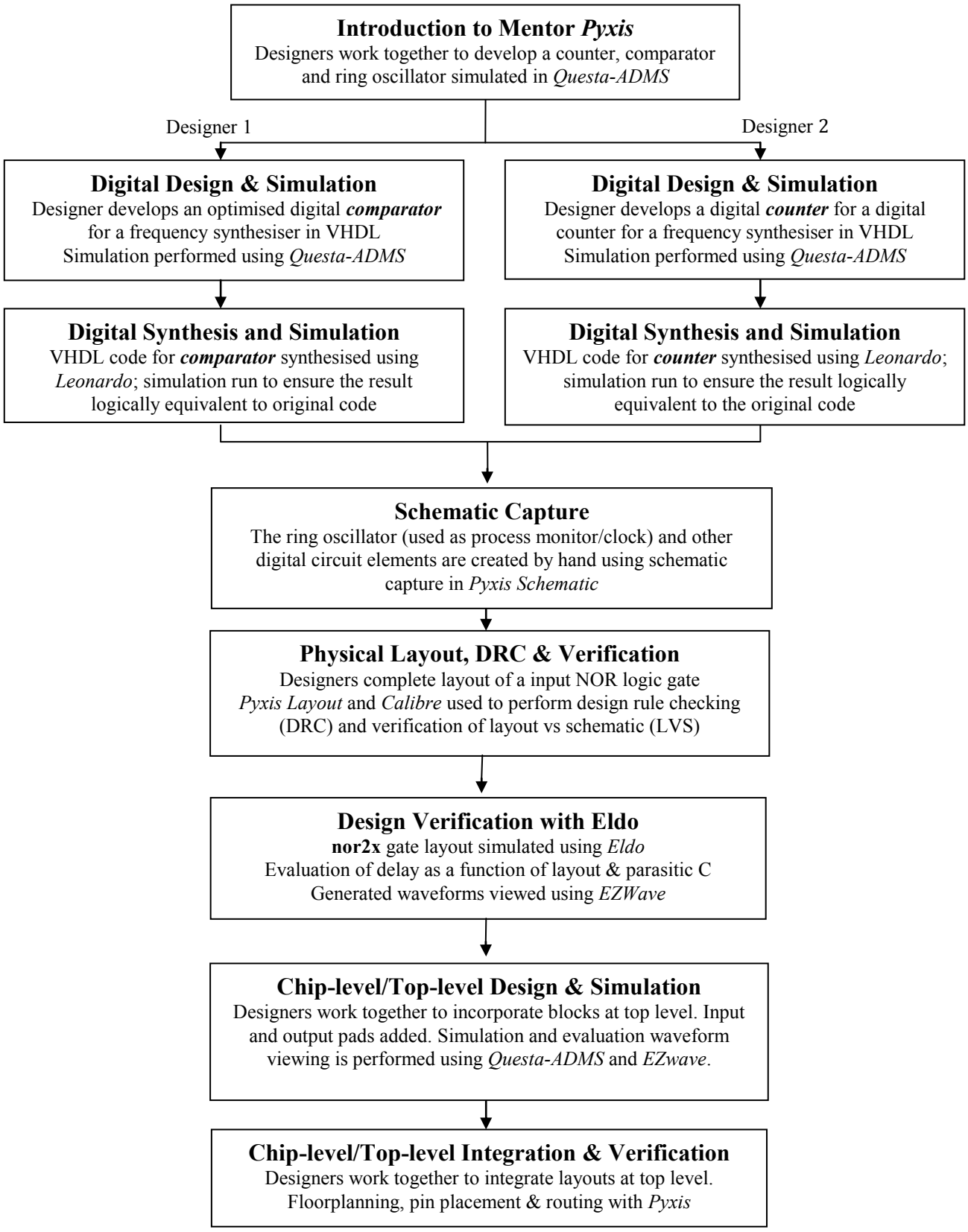
Workflow and Partitioning of the Design

Integrated circuit design is essentially a team activity because of its sheer complexity. No longer is it practicable for a single individual to take sole responsibility for a typical design. The design methodology makes intensive use of hierarchy, standards, design re-use, and sharing of knowledge among collaborators in order to keep the intrinsic complexity within bounds.

This project aims to give a flavour of as many aspects of IC design as possible, including elements of teamwork where this is expedient. The diagram on the following page sums up the key stages of the project, highlighting those parts that are carried out in collaboration, and those done individually.

Introduction to VLSI Design

Mentor Graphics *Pyxis* Design Environment
Case Study: Digital Frequency Synthesiser


Introduction to Mentor *Pyxis*
Designers work together to develop a counter, comparator and ring oscillator simulated in *Questa-ADMS*
Designer 1
Designer 2
Digital Design & Simulation
Designer develops an optimised digital ***comparator*** for a frequency synthesiser in VHDL
Simulation performed using *Questa-ADMS*
Digital Design & Simulation
Designer develops a digital ***counter*** for a digital counter for a frequency synthesiser in VHDL
Simulation performed using *Questa-ADMS*
Digital Synthesis and Simulation
VHDL code for ***comparator*** synthesised using *Leonardo*; simulation run to ensure the result logically equivalent to original code
Digital Synthesis and Simulation
VHDL code for ***counter*** synthesised using *Leonardo*; simulation run to ensure the result logically equivalent to the original code
Schematic Capture
The ring oscillator (used as process monitor/clock) and other digital circuit elements are created by hand using schematic capture in *Pyxis Schematic*
Physical Layout, DRC & Verification
Designers complete layout of a input NOR logic gate
Pyxis Layout and *Calibre* used to perform design rule checking (DRC) and verification of layout vs schematic (LVS)
Design Verification with Eldo
nor2x gate layout simulated using *Eldo*
Evaluation of delay as a function of layout & parasitic C
Generated waveforms viewed using *EZWave*
Chip-level/Top-level Design & Simulation
Designers work together to incorporate blocks at top level. Input and output pads added. Simulation and evaluation waveform viewing is performed using *Questa-ADMS* and *EZwave*.
Chip-level/Top-level Integration & Verification
Designers work together to integrate layouts at top level.
Floorplanning, pin placement & routing with *Pyxis*

Standard Project in VLSI Design SB1

Laboratory Guide 1 - a brief introduction to the Mentor Framework

This laboratory guide provides an introduction to the features of the Mentor Graphics Design Framework and the *Adobe Acrobat-based* On-Line Documentation Browser.

1. Logging in

Log in as instructed using the login assigned to you. Use the information in the *Getting Started* section and/or the *Quick Start* card to start up.

2. Start up *Pyxis* Project Manager

Most of your design and development work will be done using a new integrated development environment for chip design called *Pyxis.* This allows for management of projects, capture of schematics and design layouts, and even basic simulations can be carried out within the *Pyxis* environment. *Pyxis* supports the notion of *projects*, that is, assemblies of linked data files all related to the same design project. The first time you start Pyxis you need to set up a project to contain the files you develop. Each designer should do this individually, though note that later in the project you will each have an opportunity to work on a different part of the project, with the results being merged into a joint final design.

To start up *Pyxis,* first issue the **source** command to execute the startup script described in the *Getting Started* pamphlet. Once execution has completed you should find yourself in the **cbt** directory of the shared **mentorxx** workspace, in which projects you create will reside.

Give the following command in the terminal window:

```
dmgr_ic
```

This will start up *Pyxis* in its Project Manager mode.

3. Explore *Pyxis* Project Manager

When *Pyxis* Project Manager starts, you will be presented with a familiar form of graphical user interface with a **menu bar** at the top, one or more tool bars below, and a number of panes arranged to display information of various sorts. Most other Mentor tools present a similar interface; some have additional toolbars or palettes of commands to left and right.

The topmost pane is the **Project Navigator** window. This displays projects and other structured hierarchical data related to them. Access is by ‘point and click’. The first time you start it up there should be no projects, as you have yet to create these. In fact, only ‘open’ projects are displayed in this space – see on for more information about open or closed projects. The left half of the pane displays the hierarchy of the design space, which will include projects and libraries; the right half shows the contents of these. Typically this is the list of **design objects** within a project or library. A design object could be anything from a single transistor to a multi-million gate digital block. This approach is useful for organising your designs as reusable blocks.

A **Tools** menu**,** accessed from the menu bar, allows you to call up a number of supporting tools without leaving the Project Manager GUI. It is possible to display these in a window of their own, but this is not normally necessary.

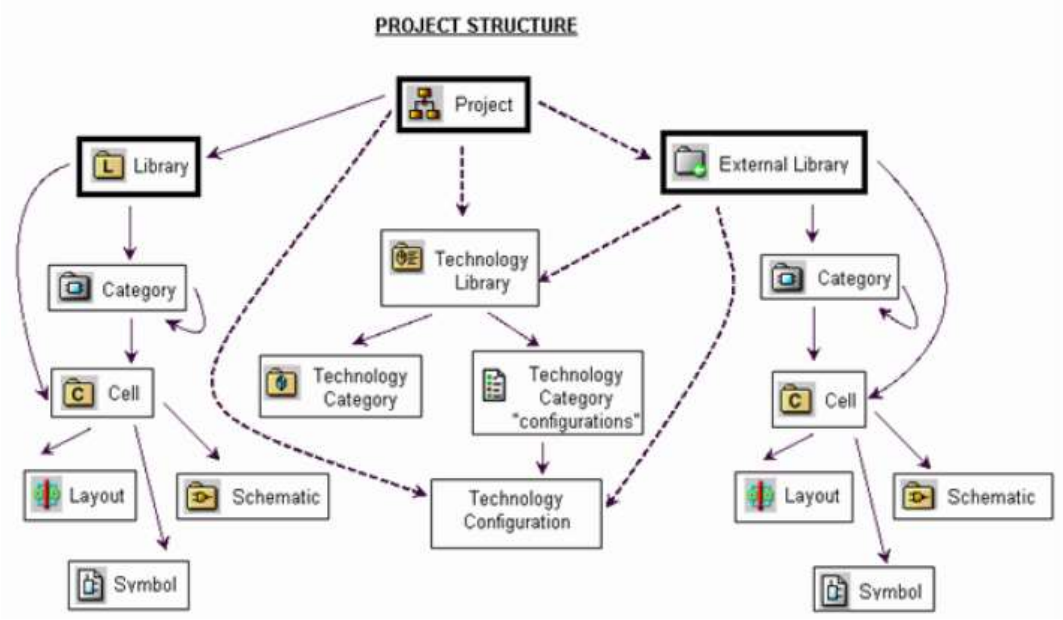
At the bottom are **Transcript** and **Message** areas. The **Transcript** area holds the complete set of commands issued during the current session, presented in an internal script language form. The **Message** area receives diagnostic or warning messages. You generally will only need to refer to this window if you have difficulty starting up other programs in the Mentor Graphics package.

Other windows may appear temporarily as you administer your projects and libraries. You can see more details in the User's Manual for *Pyxis* Project Manager – see *Accessing Mentor Documentation ...* later in this session. Feel free to explore the various menus available, although you should take care not to issue commands that will delete or change any pre-existing design data at this stage.

4. Create a project for your work

Projects are *Pyxis* Project Manager's solution for organising all data that is specifc to a design project, within a single directory hierarchy. The main components of a project are: **Project, Technology Library, Library,** and **Cell**. In addition, In addition, *Pyxis* supports the idea of an **External Library** which must be compatible with the Project, but resides outside it.

The structure of the Project hierarchy is shown in the diagram below.



- Projects, Libraries and External Libraries must reside at the root of the hierarchy
- Dotted arrow-lines represent *references;* solid arrows indicate *containment*
- The External Library references a compatible Technology Library and may contain cells

You may create as many projects as necessary within *Pyxis*. For this project, each designer should create one project, compatible with their co-designer's, to allow for interoperability of design objects.

Once created, a project's hierarchy can be kept open or closed. This is useful when there are multiple existing projects and you wish to view and work on just one at a time. It is possible to close a project by issuing the **(menu bar) File > Close Hierarchy** command, placing it in a dormant state and removing it from display in the **Project Navigator** window. However, such projects can easily be re-opened with the **(menu bar) File > Open > Hierarchy** command, and browsing to the directory where the project exists (**$CBT_WD** for this project).

If you use a regular file browser, you will be able to explore how these files are organised. However, you should never use the file browser (or commands like **cp, mv**) to make changes to any of these files. Any changes should only be made from within *Pyxis* Project Manager, which will take care of any necessary dependencies.

To create a new project for your work, follow the steps below, which are fairly intricate, but you should only need to do this once. Each designer should carry out these steps.

Give the command: **(menu bar) > File > New > Project …**

When the **Create New Project** dialogue opens, enter the prefix:

`$CBT_WD/[project-name]`

into the **Project Path** text box, followed immediately by a project name of your choosing – make it something that will be meaningful to you in the context of this project. For example: **`$CBT_WD/sb1-xyz99`**

on the assumption that one designer has the CRSid **xyz99**.

In the **Technology** section, enter the following into the **Library** text box:

`$HK_C35`

This is a soft prefix that references the AMS C35 Technology. Once you enter this, the lower part of the pane should populate with C35 process references.

OK the dialogue box and wait a few seconds – this part of the process takes a little while, as it will create a new project directory and set up various configuration files that point to the selected AMS technology.

In a few moments the **Manage External/Logic Libraries** should appear. The following steps are important to ensure your project will have access to standard-cell and other libraries. If this part is not completed correctly you will have difficulty later.

In the top row, click on the yellow folder icon. In the **File Browser** dialogue that opens up, enter the following prefix into the **Look in:** text box, clearing any other text if necessary:

`$AMS_DIR` followed by **Enter**.

This prefix points to the root of the required AMS library cells for this process. The file browser should now list a number of library folders under this prefix. Double-click on **lib**, which should open up to show a list of files of type: **mgc_ext_lib** or **mgc_logic_lib**. All are needed for this project (though some will not be required until later).

Select the first entry, **ANALOGLIB**, and OK the dialog. The top row should populate with details of the newly-imported ANALOGLIB library, in black. If you make a mistake, you can click on the faulty entry and click on **Remove Selected**, then resume. Otherwise, if the text appears in red, something has gone wrong, and you should ask a demonstrator. Assuming all is well, a second row will open up.

Click on its yellow folder icon, and repeat the process just carried out, for CORELIB and GATES. You may see other libraries listed, but they are not required at this stage[1]. On each subsequent occasion, the file browser should open at the correct directory, simplifying this operation.

Once this has been completed you should have three external library entries in the **Manage External/Logic Libraries** dialogue box. There is one further step.

Click on the button: **Add Standard Libraries**. This loads a number of generic libraries supplied by Mentor Graphics, needed for schematic entry, simulation, and other activities. Five further libraries should appear.

Assuming all has gone well, OK the **Manage External/Logic Libraries** dialogue.

Your project should now be visible in the **Project Navigator** pane. View the hierarchy by clicking on the '+' sign of any of the items, and note what appears in the right-hand part of the pane.

5. Understanding libraries

Pyxis Project Manager allows you to organise the elements of your design in a logical way in collections called *libraries*. In effect, these are customised directories stored hierarchically within your workspace. *Pyxis* Project Manager presents a graphical representation of these to assist you in accessing the data within. You can see additional details in the User's Manual for *Pyxis* Project Manager – see *Accessing Mentor Documentation ...* later in this session.

If you use a regular file browser, you will be able to explore how these files are organised. However, you should never use the file browser (or commands like **cp, mv**) to make changes to any of these files. Any changes should only be made from within *Pyxis* Project Manager, which will take care of any necessary dependencies.

Pyxis does not provide access level protection itself and depends on the underlying file system to provide it. Only one designer can work on a given entity at any one time. So care needs to be taken if you choose to work on a shared library.

6. Using libraries

In order to work together as a team, it may be useful to work on each part of your project individually, while still having access to each partner's work.

A good way to facilitate this is to create your projects in different libraries and share your libraries with each other. Each user may create local libraries with unique names, and each partner user maps the other's libraries into their own project space.

[1] The following additional libraries are provided by AMS: ESDLIB, IOLIB_4M and IOLIB_ANA_4M.

You may need to take a few minutes to plan out how you will be organising your work.

7. Creating a custom library

You can create a library within your own project to help organise your design objects. Note that it should be possible to agree with your partner/co-designer the name of a common shared library. For the purpose of this handout, for future reference, we will just call any such library the **Shared-Library**.

To do this, highlight the project you created in step 4, by clicking it once in the *Pyxis* **Project Navigator** window. **Note**: if you do not carry out this step, you will only be able to create an External Library. Give the command **(menu bar) > File > New > Library**, and enter the library name; OK the dialogue box.

Nothing further is required, as the new library will inherit references to the AMS technology from the project. Click **OK** to create the library. An internal library will appear in the Navigator pane as a plain yellow folder with an 'L' superposed.

Although it is in theory possible to rename a library any time, this is not generally advisable, as any change of name mid-design may mean that dependencies may need to be resolved, one by one. It is best to select a name and stick to it throughout your project. Your name should also be unique.

8. Accessing an external custom library

External Libraries are *Pyxis* Project Manager's solution for organising re-usable design data so it can be imported into one of more projects. An External Library must reference a Technology Library. Its contents can be used only in projects that reference the same Technology Library.

You are unlikely to need to create an External Library for this project, though ths can easily be done if necessary. Details are available in the Moodle **VLSI Design 2015** Wiki.

A certain amount of pre-existing data will be made available to designers through the medium of an External Library, **sb1_nor2_lib**, which has been set up with the $HK_C35 technology. An external library will appear outside the project hierarchy as a grey folder with an arrow superposed.

With *Pyxis* Project Navigator, navigate to the **sb1_nor2_lib** object, and click on its '+' sign to view the contents. This external library contains schematic, layout and symbol objects which you will need in a later session.

9. Cut/Copy/Paste/Delete

It is possible to move designs around by using the Cut/Copy/Paste/Delete commands. It is possible to perform these operations on libraries, groups, cells and views. Try this out by copying an existing technology library cell into your custom library.

Care should be taken when deleting any part of your design, as it is not normally recoverable.

10. Access Mentor documentation using Acrobat or a browser

We hope that the majority of the information you will need for this project can be found in this Project Guide. However, there may be occasions when you need to consult the official documentation for the packages.

Mentor documentation is held in two different forms to suit different requirement: Adobe PDF, and html. Instructions are given for both, and both are likely to be useful in this project.

To open the html documentation, select the item:

(**menu bar) Help > Open User's Manual** or alternatively

(**menu bar) Help > Open Reference Manual**

to start up the default browser, normally Firefox, at the cover page of the compilation. A pane at left allows navigation from chapter to chapter, and tabs in that pane give access to an alphabetical index, and to a 'search this document' feature. An Acrobat icon at top right allows you to select the Acrobat version of the reference, normally by download to your own workspace; the Acrobat utility can then be used to view the document. Another button gives details of other related manuals that can be accessed.

Sometimes it may be more convenient to browse PDF manuals outside the Pyxis environment and even away from the workstation. Although printed copies of the manuals are not available (they extend to many tens of thousands of pages), most of the key manuals can be accessed on a computer within the University by means of a web browser. **Please note that it is a condition of our End-User Agreement with Europractice, Mentor Graphics and Austriamicrosystems that we may not distribute this documentation except to those involved in these projects, and copies may not be made.**

11. Accessing AMS 0.35μm CMOS process documentation

A copy of parts of the original AMS documentation for the 0.35μm process, cell library and other features is available locally, in PDF form, via the **Moodle** support platform. The **Moodle** support for this project is described in step 12, so you may prefer to carry out this investigation as part of that step.

To access the documentation, start up a suitable web browser and use it to access **Moodle** (see step 12) and open up **SB1: VLSI Design**.

In the main menu, look for **Project Library** and click on **Project Documents**. Scroll to the section: **Mentor and AMS User Manuals and Reference Guides**.

The **AMS 0.35 μm CMOS Digital Standard Cell Databook** describes the characteristics of the digital cells, from inverters to complex logic systems, available in the AMS C35 library. It is a useful reference for as designer planning to use AMS library cells in a design. This may be useful in Labs 3, 4, 7 and 8.

The **AMS C35 0.35 μm CMOS Design Rules** reference describes the physical and electrical design rules that apply to designs for manufacture by AMS. Although design-rule checking is undertaken by the design tools themselves, correcting any violations inevitable requires that the designer should understand the way the rules are meant to apply. This may be useful in Lab 5.

12. A quick tour of the Moodle support platform

There is a dedicated **Moodle** Support Platform in support of this project. This is accessible 24/7 from any workstation or networked PC with a suitable browser. It is open to all participants, demonstrators and staff, and offers a convenient way of sharing information among participants, including outside the timetabled sessions. We strongly recommend that while engaged in the project you should get into the habit of checking **Moodle** on a regular basis, since it will be used routinely for speedy distribution of information e.g. reminders about deadlines, minor amendments to the lab sheets, etc. Start up the *Mozilla Firefox* application during your session so you can monitor it. This is especially important outside scheduled periods, when the demonstrators will not be available

This procedure guides you through a quick tour of the main features of the worksite.

Start a suitable web browser; for the Teaching System this will be Mozilla Firefox, and navigate to the **Moodle** entry point:

`https://www.vle.cam.ac.uk/`

You will be asked to login with your Raven ID and password.

On your **Startpage** you should see (in the **My Course and Projects** pane) an entry: **SB1: VLSI Design**.

Click on the link and you should be taken to the project **Home Page**.

At the foot of the page, below the heading, is a short count-down message indicating how much time remains to the next key deadline e.g. start of project, hand-in of next report, etc. This is updated each time you visit the home page.

The main menu in the centre contains links to the main features of the worksite. There is a summary table of the most important of these at the end of this section. All sections are accessed by clicking the links in the left-most column. We recommend you briefly visit them all, to get some familiarity with the service they provide. You are also welcome to visit the additional links not mentioned below.

First visit the **Document Library / Project Documents** feature.

Here are held electronic copies of the Project Guide, as a single PDF, and section by section. If it becomes necessary to update any of the Lab Guides, they will be published here as individual documents. The complete guide will not be updated.

Further down, you can find links to the most important manuals in PDF form for the Mentor Graphics tools in use for this project. Access to these does not depend on you being logged in at the Mentor server, **clae01** or **clae02**.

The links are to a site which allows access from workstations in the .cam domain – note that the CamTools site itself may be viewed from anywhere. This is necessary because of restrictions imposed by the companies concerned.

Now click on **Announcements**. These fulfil the same function as on any **Moodle** site. If there are any high-priority announcements, you should also receive details to your personal email account.

Now click on **Discussion**. The Discussion Board is divided into a number of relevant sessions, roughly mapped on to the various sections of the Project. If you have difficulties when undertaking this project, particularly if they occur outside the scheduled periods, we strongly recommend you post a question (with as much information as possible about the specific problem). The message will be seen by all users and demonstrators, and there is a far greater probability that someone will come up with a helpful suggestion than if you were to try email. Before posting a message, scan the existing posts and the Wiki – see below. Someone may already have encountered the same problem and may have posted a solution.

Members of the project are welcome to use the **Chat Room** for live exchanges, particularly out of hours. The demonstrators and others monitor the Chat Room, and will be able to chip in if they can help.

The **Wiki** is organised in the form of an FAQ, and currently has about 40 entries. A few may need update as there have been radical changes to the project this year. You are welcome to contribute amendments or updates yourself if you become aware of an error or of a better solution to a problem.

The **SB1 Signup** feature (*still under development, and may not be published until after the project starts*) lists all the available slots that can be booked for the accompanying experimental activity – **an investigation of the electrical characteristics of a ring oscillator** – which occurs in weeks 2 and 3 of the project, during normal scheduled sessions. We can accommodate a maximum of 6 students per 2 hour session, so about three two hour sessions will be needed. The **signup** tool will gives you the opportunity to pick a time when there is a natural break in your design work. You will be able to work in pairs for the experimental activity.

Reminder: if you run into problems outside the scheduled sessions, please post details of the difficulty to the Moodle Discussion Board. This is monitored by demonstrators on a regular basis, and while we cannot guarantee turn-around on solutions for problems reported in this way, we will always do our best.

This concludes our preliminary investigation of *Pyxis*, the **Mentor** environment and the **Moodle** support platform.

Moodle Support Platform Summary Table

Please remember you are welcome to access **Moodle** from your own account at any time and from anywhere you have web access. The URL is as follows:

Worksite: `https://www.vle.cam.ac.uk/` **Title:** `SB1: VLSI Design`

Part IIA Project: SB1: VLSI Design	The Home Page; description of project and worksite; actively updated key dates
Project Library	Links to PDF copies of the Project Guide, Manuals for Mentor tools, AMS references, other reference material
Calendar	A calendar listing scheduled sessions in the DPO, availability sessions for the linked activity in EIETL, and report deadlines
Announcements	Information about updates, amendments to documents, scheduling, etc
Discussion Board	The VLSI Design Forum. This is where you should post any questions or comments on difficulties you may be having, so all demonstrators and users have the benefit.
Chat Room	Live chat with live IC designers ;-) [availability TBC]
Wiki	This is organised as an FAQ, and is updated and extended regularly
SB1 Lab Signup	Sign-up facility for the lab activity in EIETL [TBC]

Standard Project in VLSI Design SB1

Hardware Description Language VHDL

This pamphlet briefly introduces the hardware description language VHDL and its basic characteristics, and describes the evolving role of this technique in the specification and verification of digital circuit designs.

The traditional approach to integrated circuit design involves the capture of a schematic - a collection of symbols and wires that expresses the structure of the design. While schematics are useful in allowing the designer to have access to a graphical picture of the design, a serious shortcoming of the approach is that it is difficult to embody the *behaviour* of the design in a schematic. Hence the verification of the correctness of a design by examining schematics is at best an indirect process.

Hardware description languages (HDL) can replace circuit diagrams, flowcharts and other documents as the primary documentation medium for circuit structure and behaviour. They are especially well suited to the design verification process, because, with appropriate supporting software, an HDL description can be directly executed to simulate the circuit it describes. HDLs also provide a convenient way to describe the input/output data used by logic synthesis tools.

The value of a description language can also be illustrated in a different way: suppose you have designed a 12-hour digital clock and you now wish to redesign it as a military-style 24-hour clock. In the schematic form of digital systems implementation, you would need to redesign your clock schematic substantially, changing and re-specifying gates and rewiring the design. With a design described in a hardware description language, you could simply revise your specification, changing 12 to 24 in a number of places, verify the description on a computer to check it behaves as expected, and use the result as input to an automated semi-custom layout system similar to that available with the Mentor Graphics suite.

An attractive approach for specifying behavioural descriptions is through a *hardware description language*. These languages look much like conventional high-level computer programming languages and contain many of the constructs of languages like C or Pascal. However, conventional programming languages rather compel the user to think of executing a single statement of the program at a time. This is unsuitable for hardware, which is inherently parallel: all gates are constantly sampling their inputs and producing new outputs. The high degree of parallelism in most hardware is one of the things that makes hardware design and verification difficult.

VHDL (VHSIC Hardware Description Language) is a widely used language for hardware description, and is based on the programming language ADA. We do not attempt to present VHDL in anything like its complete form here, but the notes below may hint at the power of the technique.

A piece of digital hardware at any complexity level from an entire system to a primitive gate is called a **design entity**, and is specified by a composite VHDL statement of the form:

entity *Name* **is . . . end;**

Figure 1 begins with an **entity** statement that identifies the hardware item of interest as *mux*, a user-specified name, and lists its primary input/output signals and their

types. The input/output signal list is introduced by the keyword **port** (a term often used to refer to a group of associated input/output lines in a digital system). The signal direction **in** or **out** is indicated in the **port** statement, as well as the signal type (in this case **bit**, meaning that the named signals assume the binary values 0 and 1 only). Alternative signal types might include an unknown value X, or we might wish to treat the signals as integers, in which case the keyword **bit** would be replaced in the **port** statement by the keyword **integer**.

In Figure 1, both *structural* and *behavioural* descriptions of the design are included. The structure and behaviour of the target circuit are described within a compund VHDL statement of the form:

architecture *Name* **is begin . . . end;**

which in Figure 1 follows the **entity** statement. Note that timing information such as component delays and signal rise and fall times can also be included in an **architecture** specification.

A basic VHDL construct is a **signal assignment** such as:=

Z <= not A;

which states the the signal Z is assigned the current value of the expression on the right of the assignment. To specify a timing delay, the keyword **after** may be used, thus:

Z <= not A **after** 2ns;

The familiar **if-then-else** construction can be used to add further logical conditions to a signal assignment: for instance:

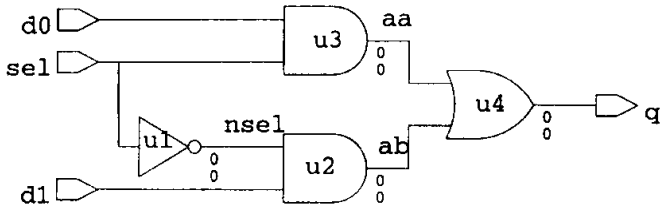
if M = '0' **then** Z <= **not** A **after** 2 ns;
else Z <= A **or** (**not** B) **after** 2.8 ns; **end if**;

Here M serves as a control signal, its value 0 or 1 determining the logical operation used to calculate the value of the data signal Z. When a large number of control conditions exist, a sequence of **if** statements can be replaced by a more concise **case** statement:

case C **is**
when C_1 => Z <= f_1(X);
when C_2 => Z <= f_2(X);
⋮
when C_n => Z <= f_n(X);
end case;

Other standard language features include the use of pre-defined functions and the association of types like **bit** and **integer** with signals and other variables.

The following section is a convenient summary of VHDL command syntax.



Gate-Level Representation of Two-input Multiplexer

```
0   -- Code of Structural Description for a Multiplexer
1   ENTITY mux IS      -- entity declaration
2       PORT (dO, dl, sel: IN bit; q: OUT bit); --port clause
3   END mux;
4
5                  -- architecture body
6   ARCHITECTURE struct OF mux IS
7       COMPONENT and2 -- architecture declaration
8           PORT(A, b: IN bit; C: OUT bit);
9       END COMPONENT;
10      COMPONENT or2
11          PORT(A, b: IN bit; c: OUT bit);
12      END COMPONENT;
13      COMPONENT inv
14          PORT (a: IN bit; c: OUT bit);
15      END COMPONENT;
16
17      SIGNAL aa, ab, nsel: bit;--signal declaration
18
19      FOR ul : inv      USE ENTITY WORK.invrt(behav);    config.
20      FOR u2,u3: and2   USE ENTITY WORK.and_gt(dflw);    specif.
21      FOR u4 : or2      USE ENTITY WORK.or_gt(archl);
22
23  BEGIN
24      ul:inv     PORT MAP(sel, nsel);--architecture statement part
25      u2:and2    PORT MAP(nsel,dl,ab);
26      u3:and2    PORT MAP(DO, sel,aa);
27      u4:or2     PORT MAP(aa, ab, q);
28  END struct;
```

```
0   -- Code of Behavioral Description for a Multiplexer
1   ENTITY mux IS                -- entity declaration
2       PORT (dO, dl, sel: IN bit; q: OUT bit); --port clause
3   END mux;
4                                -- architecture body
5   ARCHITECTURE behav OF mux IS
6   BEGIN
7       fl:                   -- process statement
8       PROCESS (dO, dl, sel) sensitivity list
9       BEGIN
10          IF sel = '0' THEN -- process statement part
11              q <= dl;
12          ELSE
13              q <= d0;
14          END IF;
15      END PROCESS fl;
16  END behav;
```

Figure 1

VHDL Command Summary

Concurrent Statements

block_statement

label: **block** [(guard_expression)]

[generic]
[ports]
[declarations]

begin

concurrent_statements

end block [label];

component_instantiation_statement

label : name [**generic map** (map)] [**port map** (signals)];

concurrent_assertion_statement

assert condition

[**report** string_expression]
[**severity** {**NOTE | WARNING | ERROR | FAILURE**}];

concurrent_procedure_call

[label :] procedure_name [(parameters)];

concurrent_signal_assignment_statement

[label :] [{conditional_assignment | assignment | selected_assignment}];

generate_statement

label : [{ **for** specification | **if** condition }] **generate**

concurrent_statements

end generate [label];

process_statement

[label :] **process** [(sensitivity_list)]

[variable_declaration]
[type_declaration]
[subprogram_declaration]
[declarations]

begin

sequential_statements -- Cannot contain a **wait** statement if sensitivity_list is used

end process [label];

Sequential Statements

assertion_statement

assert condition -- When condition is false [strng_expression] is printed

[**report** string_expression]
[**severity** { **NOTE | WARNING | ERROR | FAILURE** }];

case_statement

case expression **is --** Avoid the use of parenthesis, if possible
when choices_1 => sequential_statements
.
.
when choices_n => sequential_statements
end case;

exit_statement

exit [label] [**when** condition];

if_statement

if condition **then**

sequential_statements
{ **elsif** condition then sequential_statements }
[**else** sequential_statements]

end if;

loop_statement

[label :] [**while** condition | **for** loop_specification] **loop**
sequential_statements
end loop [label] **;**

next_statement

next [label] [**when** condition]**;**

null_statement

null;

procedure_call_statement

procedure_name [(parameters)] **;**

return_statement

return expression; --For use in a Function
return; --For use in a Procedure

signal_assignment_statement

target **<=** expression [**after** time_expression]
.
.
{ **,** expression [**after** time_expression]**;**

variable_assignment_statement

target **:=** expression **;**

wait_statement

wait --A Function may not contain a wait_statement

[**on** signal_name {, signal_name}]
[**until** conditional_expression]
[**for** time_expression] **;**

Specifications

attribute_specification

attribute attribute_name
of entity_name is expression **;**

configuration_specifications

for component_name
use [generic_map_part]

[port_map_part] **;**

Library & Use Clause

library_clause

LIBRARY names **;**

use_clause

USE selected_names **;**

Declarations

alias_declaration

alias name1 : type [(indexes)] **is** name2 [(indexes)] **;**

attribute_declaration

attribute name **:** type **;**

component_declaration

component identifier :
[**generic** (generic_list) **;**]
[**port** (port_list) ;]
end component ;

constant_declaration

constant name **:** type **:=** expression **;**
constant name **:** array_type [(indexes)] **:=** expression **;**

file_declaration

file name **:** type **is** [mode] logical_name **;**

signal_declaration

signal names **:** type [constraint] [**:=** expression] **;**

port_declaration

port (names **:** direction type [**:=** expression] [; more_signals]);

subprogram_declaration

{ **procedure** name [(parameters)] | **function** name [(parameters)] **return** type; }

sub_program_body

is

declarations

begin

sequential_declarations

end [name] ;

subtype_declaration

subtype name **is** [resolution_function] type [constraint] ;

type_declaration

type name **is** definition**;**

variable_declaration

variable names **:** type [constraint] [**:=** expression] **;**

Library Units

architecture_body

architecture name **of**
entity_name **is**

[types]
[constants]
[signals]
[subprograms]
[other declarations]

begin

concurrent_statements

end [name];

configuration_declaration

configuration name **of**
entity_name **is**

declarative_part
block_configuration

end [name];

entity_declaration

entity name **is**

[generics] [ports]
[declarations]

[**begin** statements] --typically, an entity does not have statements. If it does, the statements cannot operate on signals
end name;

package_body

package body name **is**

[subprogram]
[type]
[constant
[signal]
[declarations]

end [name];

package_declaration

package name **is**

[subprogram]
[type]
[constant]
[signal]
[file]
[alias]
[USE clause]
[declarations]
end [name];

Standard Project in VLSI Design SB1

Laboratory Guide 2 - Functional HDL Simulation with ADMS

This laboratory guide provides an introduction to functional simulation using sets of abstract models – hardware descriptions – so as to allow a clear understanding of the target design before any commitment needs to be made to detailed design. The session will introducs the features of the VHDL hardware description language, as well as the use of Mentor Graphics *ADMS* compiler and simulator suite, which incorporates comprehensive facilities for VHDL.

Questa ADMS is Mentor Graphics' mixed-signal simulation suite. It offers designers a comprehensive environment for verifying complex analog/mixed-signal System-on-Chip designs. *ADMS* combines four high performance simulation engines in a single tool: *Eldo*® for general purpose analog simulations, *Questa*® for digital simulations, *ADiT*™ for fast transistor-level simulations and *Eldo RF* for modulated steady state simulation. ADMS is language neutral; you can combine VHDL, Verilog, VHDL-AMS[1], Verilog-AMS, SystemVerilog, SPICE and SystemC (C programming code) anywhere and at any level in the design. *Questa ADMS* includes the *EZwave* mixed-signal waveform viewer and waveform calculation tool for display and analysis of mixed-signal results. The broad range of applications served by *Questa ADMS* means that the setup for VHDL-only is rather more elaborate than might be the case with a simpler, stand-alone VHDL modelling tool, but the extra complexity is amply compensated by the added flexibility

The uses of VHDL in digital systems design have been described in another pamphlet. In this session we shall begin by using VHDL to express the structure of a simple counter, followed by the ring oscillator, which elements will later form the heart of this project. In addition, we will be able to explore a behavioural model for the two-input NOR gate used in its construction. You will see how a wide range of digital hardware can be modelled in terms of statements similar to those in computer programs, e.g. Pascal or C. These models will then be simulated using *ADMS*, but under control of the *Pyxis Language Interface*, primarily to demonstrate the close coupling these tools offer. Once the operation of the basic counter and ring oscillator have been explored, it will be possible to carry out a more detailed investigation of the target design, viz. the programmable counter, comparator and associated logic required in a frequency synthesiser. This will be done using *Questa ADMS* in a stand-alone mode.

At this stage of the design process we may have made no decisions about the way in which the hardware will ultimately be implemented, but by use of models of this kind we can gain considerable insight into the way the design will work and explore a number of 'what if' scenarios – for example, the effect of different gate propagation delays for rising and falling edges. Later in the project, if time permits, we shall substitute into the VHDL model the parameters corresponding to the delays for the actual NOR2 gate which you yourself will design in week 3.

[1] Note that AMS stands in this context for Analogue and Mixed Signals and does not relate to the Austriamicrosystems (AMS) technology used in the project.

This approach will later help us synthesise a design based on standard logic gates from the Austriamicrosystems Cell Library. This is a very useful feature, especially for big designs. However, the great strength of VHDL and other HDLs is that they allow us to think and plan in abstract terms without necessarily having to commit to a specific technology.

Before you start HDL modelling, you will need to give thought to the form of the ring oscillator module. You can do this off-line, away from the workstation. You should review the material in the introductory document **Design of logic gates in CMOS**, together with data in the **Design Specification** pamphlet, especially the specification for the ring oscillator itself. You may find it helpful to start with a schematic sketch of the ring oscillator. The number of gates required will be dictated by the delays and oscillation period specified in the **Design Specification**.

You should be logged in at a workstation as described in the **Getting Started** section, with *Pyxis Project Manager* open.

In this section we shall initially examine straight-forward designs using the ring oscillator (based on the 2-input NOR gate) and a standard counter, COUNT-EVAL, in order to develop experience in this environment and to help evaluate the ring oscillator concept; later we shall progressively expand and adapt the design to incorporate additional features and modify it so that it meets the specification set out in the Design Specification section above.

1. Examine the resource directory `sb1_hdl`

Open a terminal window and navigate to `$MHOME/sb1_hdl/sample`. A text file `sample.vhd` containing a number of VHDL sources for a 2-input NOR gate, counters, and other elements is provided. You will need to edit and adapt these to model your proposed design, using copy-paste. The file is not suitable for direct compilation.

The built-in editor for *Pyxis*' Language Interface is functional, but spartan. While it is satisfactory for small edits or corrections, you may wish to use the Linux editor *gedit* so you can prepare documents in another terminal window.

2. Create a new VHDL file

Using the *gedit* editor, or your preferred alternative, develop a new `.vhd` file with your own choice of name – for example, **counter.vhd**. This should contain VHDL code to implement (only) the simple counter presented at the start of `sample.vhd`. Save it in the `$MHOME/sb1_hdl` directory.

Many aspects of VHDL coding are beyond the scope of this handout. For information about the background to HDL coding, and for tutorials, look in the resources found on the **Moodle** support platform **SB1: VLSI Design.** Illustrative code based on `sample.vhd` is printed for reference at the end of this section – but note that it will require *careful review and adaptation to your specific needs*.

For the purposes of this introductory section, you should develop code that will implement a working 4-bit VHDL counter – but note that this will not satisfy the needs for your final design.

Note that **counter.vhd** represents a test counter intended for use in evaluation of the ring oscillator. Later on you will need to determine a more suitable specification, and will need to dedicate some further thought to this. You will also need a more appropriate name for the counter/s needed in your final design. Be careful to use names that will not cause clashes later on.

3. Editing the VHDL source code

The following paragraphs give advice on editing the file to develop suitable code. You should refer to them in later sections when you develop other parts of the design.

Referring to the various resources mentioned, enter suitable code into the text window; this will be stored in your **$MHOME/sb1_hdl** directory.

Pay particular attention to the **USE** section at the top of each section of the code. If the proper libraries are not included in the code, the code may compile and simulate correctly but will not synthesise properly in later stages of the project. In particular, you will need the following libraries:

IEEE.STD_LOGIC_1164.ALL
IEEE.STD_LOGIC_ARITH.ALL
IEEE.STD_LOGIC_SIGNED.ALL

For compatibility with the standard cell libraries that we will use for this project, we **strongly recommend** that you use the following logic conventions for your digital design:

Clock	: positive edge
Reset	: asynchronous active low
Enable	: synchronous active high

Ask a demonstrator if you are not sure what this means.

You should also name your bus vectors in ascending order, from 0 to N, to correctly plot the outputs in the proper order during simulation. Otherwise, the output order may be reversed during simulation. [This is a problem with the *EZWave* graphic display program, which does not provide any easy way to re-order the signals].

Note: Please make sure that the **`ENTITY`** names used in your **VHDL** code (note: these are not the file names) end with the string: **`_vhdl`**, which should be the case if you retain the original names suggested. This will help ensure that the cells will be correctly renamed during synthesis into a CMOS schematic design, to be undertaken later.

4. Open the Pyxis Language Interface

With your project highlighted in the Project Navigator Window (single click to select if not), click on **File > Open > Language Interface**. A new form of Navigator screen - *Pyxis Language Interface* - should appear. Check that the left part displays the hierarchy of your project. This navigator does not permit you to make changes to your project structure; its role is purely to allow you to introduce resources from outside the project.

5. Import the counter VHDL model to your library

Within the Language Interface, and with your new library highlighted (left-click once if necessary) in the Project Area pane, give the command: **File > Import > HDL**. The *Import HDL* dialogue should open. Using the file navigator for the *HDL Source* item, navigate to the vhdl directory, and select the `.vhd` file for your evaluation counter model. With this done, the remaining fields should fill out automatically. Edit the *Import Source To Cell* field so that the last part indicates an appropriate name of your choice – you can leave it unchanged if you wish. This cell will receive the contents of the `.vhd` file. *Pyxis* will create an additional cell based on the **ENTITY** name and prepare a symbol ready for you to instantiate in a test-bench design. Details should appear in the **Registration** section of the dialogue. Once you are happy that this is correctly set up, OK the dialogue to launch the import, which may take several seconds to execute and register the new model/s. Assuming no error messages appear in the Message Area, you should now be able to identify the newly imported HDL file and a new cell containing a symbol representing it, which can be instantiated in any design that requires it..

6. Compile the VHDL code into a simulation model

With the Language interface still open, select your project in the Project Area of the Language Interface; give the command **Edit** > **Compile**. This operation will take a while and unless your coding is perfect and free from typographical and other errors, is liable to result in compilation errors. Watch the Message Area at the foot of the screen, which will display a path to a transcript file of the form: `/tmp/lmrc_transcriptnnnn`. You should open this file with *gedit* or a similar tool to identify any compilation errors. If there are errors, you may find it helpful to select the VHDL source file within your library and give the **File** > **Open** command to see it within *Notepad* in the *Pyxis* environment. Give the command **View** > **Show Line Numbers** to help you identify where the errors lie.

Check your code and make any changes and give the command: **File** > **Save**. Recompile and monitor the diagnostics, which should appear in the same transcript file. Repeat as required until the code compiles successfully. This is essential to ensure that the design will simulate under *Questa ADMS*.

Once all necessary code has compiled satisfactorily you no longer need the Language Interface. Close it by giving the command: **MGC** > **Exit**. You may wish to right-click in your project hierarchy and give the command **(RMB)** > **Refresh**, to ensure that *Pyxis* Project Manager displays the most up to date information.

7. Re-editing the VHDL code

If you wish to edit your VHDL code at any other time, you can do so by selecting the appropriate **Cell** in the Project Manager and double-clicking on the **VHDL view** to open it in *Notepad*. *After making any changes, you should always check your code*. Use the Language Interface as described above to access the **Compile** feature.

Note that if you edit the VHDL code in such a way as to change the interface (e.g. add one or more new signals), the symbol will not be automatically updated. **If in doubt**, delete the symbol and re-import the model as described in step 5 above.

Designing a test schematic

8. Create a new test schematic for the counter

We shall now design a simple test schematic to perform functional simulation. With the library containing the counter cell highlighted, give the command **File > New > Schematic** and enter the following information in the **New Schematic** dialogue:

Cell Name : counter_testbench
Schematic Name : (leave unchanged)

Click OK to open the new schematic with *Pyxis Schematic*. *Pyxis Schematic* is basically a drawing tool and behaves quite similarly to many common graphical design editors.

9. Insert a single instance of the counter

Select **Add > Instance > Choose Symbol** and a dialogue will allow you to choose which symbol to add. Navigate to the newly-imported **counter** component and select the **count4_vhdl** symbol within it; click **OK** to select it for placement.

You are now in *place mode*. Your cursor will change and you should see the outline of the design symbol under it. You will also see a mini dialogue appear at the bottom of the schematic window.

Move the design to where you want it to be on the schematic and left-click to drop it in.

10. Insert input and output ports

Insert input and output ports using the same technique. You can find these under the following Cells:

Input Port : $MGC_IC_GENERIC_LIB/portin/portin
Output Port : $MGC_IC_GENERIC_LIB/portout/portout

Drop them onto the schematic close to the input and output pins of the counter symbol. Instead of repeating the whole process for each input/output, you can also copy/paste the repeated symbols instead.

11. Wire the Nets

Once all the symbols are placed, select **Add > Wire** to enter wiring mode. This is indicated by the appearance of a mini dialogue near the bottom of the schematic window. The mouse cursor will also change to a cross.

Click on the starting point and double-click on the end point for each wire. To make bends, click on each point of the bend. Once you have completed wiring the schematic, click **Cancel** on the mini dialogue to exit the mode.

12. Check the schematic

You should now check the schematic. This will check the schematic for any drawing violations. Select **File > Check Schematic** to do this. A report window will appear with a list of errors. At this point, you should see a number of **Net Errors** being reported. Select **File > Close Report** to close the report and return to the schematic window.

13. Name the Nets

To fix the errors, we will need to give each Net a unique name. First of all, deselect any part of the schematic that has been selected by clicking on an empty part of the schematic (or press F2). Then, select the specific wire to name by clicking on the wire.

Select **Edit > Edit Object** to enable the **Edit Object** dialogue. You can alternatively use short-cut key: **Q**. Enter the following information in the row with Attribute displayed as: Net Name

Value : <New Net Name> (e.g. CLK, RST, ENA)

Click **OK** to confirm the change. You should see the name of the input or output port change to reflect its new name. Repeat this for all the other wires. To name a bus or wire bundle, you will need to specify a range in the name. Repeat the above for the counter outputs and enter the following:

Value : COUNT[0:3]

Be sure to include the square braces and correctly specify the size of the bus or bundle. [0:3] signifies a 4-bit bus. Click **OK** to confirm.

14. Check and save the schematic

Once all the Nets have been renamed, check the schematic again using the same technique above. At this point, there should not be any errors in the report. If there are still any errors, you should fix them before continuing.

To save the sheet, select **File > Save Sheet** and *Pyxis Schematic* will save a new revision of the sheet. It will report this in the status bar.

Do not close *Pyxis Schematic* yet as you will still need to use it for simulation. If you have closed it, you can open it up again by double-clicking on the schematic view of the cell you just created from within *Pyxis Project Manager*.

15. Enter simulation mode

Pyxis Schematic can also be used to invoke the simulation process. You can enter this mode by clicking on the green arrow at the bottom of the left toolbar. A dialogue **Entering Simulation Mode** will pop up to ask for the **Configuration** to use. *Questa ADMS* is capable of several types of simulations, and more than one approach may be needed in a single design As a result, the dialogue boxes and commands needed have to serve multiple purposes and the procedure required is slightly more complicated as a result. For convenience, click on **New Configuration**, and in the resulting **New Design Configuration** dialogue box, enter a suitable name – for example, **VHDL.** Select the **ADMS** radio button and click OK. There is no need to change any of the **Options** at this stage.

You will be able to use this new configuration to invoke VHDL simulation results in later sections.

Click **OK** to enter simulation mode. *Pyxis Schematic* will visibly change after entering this mode. The left toolbar will reformat to display links to simulation tools in place of drawing tools. The most useful; of these include: **Setup Simulation …**, **Run Simulator**, and **Plot Results from Latest Run**. The schematic window will shrink to make use for other menus.

First, ensure that the simulation tool palette is visible on the right, by selecting **Setup** > **Windows** > **Palette Area** (which should then display a tick mark). You will need some of the tools in the palette to carry out the next few steps – though as with many of these tools, there are usually alternative ways of achieving the same result.

Running functional simulation in ADMS

16. Set up Simulation

ADMS integrates directly with Mentor Graphics' *Pyxis Schematic*; invocation and control of a simulation can be carried out without leaving *Pyxis Schematic*. Although this is intended as a functional simulation, *ADMS* expects signals to be defined in terms of voltages, and steps need to be taken to map voltages on to logic levels. This is handed by the provision of *Data Converters* at every signal input or output. Note that these are virtual devices whose only role is to allow the simulation tool to interconvert between standards, and they do not imply additional hardware. The specification of these converters is important in ensuring that the simulation runs smoothly. If you want to know more, see: *Questa ADMS* Command Reference, ADMS 12.1, p199-208, which you can access from CamTools.

Preparation for simulation is carried out in the **Setup Simulation** dialogue. This can be accessed in more than one way. Either:

- from the tool palette, select **Analysis …** , or
- click on the **Setup Simulation** icon (ac, dc, tran) in the left toolbar.

With the Setup Simulation dialogue displayed, check and verify the following, making changes as necessary.

(a) With **Analysis** highlighted in the menu at left, examine the **Analysis Selector** panel, and ensure the **OP** and **TRAN** features are both checked. On the right, make sure **Enable TRAN** is checked, and if necessary, set a suitable **Stop Time**; the default of 1000n (1000 ns) will be satisfactory.

(b) With **Converters** highlighted in the menu at left, verify the following:

The **Auto** check box should be checked, and the **Builtin** radio button should be checked.

With the **A2D** (Analogue to Digital) radio button set, do the following:

Select **Model** = **Std_Logic**; click **Add**. No other parameters need to be changed. Click **Add**.

With the **D2A** (Digital to Analogue) radio button set, do the following:

Select **Model** = **Std_Logic**; click **Add**. No other parameters need to be changed. Click **Add**.

Check that as a result of the above actions, two enabled converter boundary models have appeared in the report panel at the foot of the dialogue box. If all is well, click **Apply**, then close the dialog box, and return to viewing the test-bench schematic for **count4_vhdl**.

17. Add a fixed signal or force

Before we can simulate anything, some of the input signals will need to be defined. There are several ways to set up these signals. In the ADMS environment, inpout signals are referred to as **Forces**, and although in this instance it is intended that they be thought of as logical values, i.e. **0** or **1**, they have to be set up as voltages.

The enable signal for the counter needs to be fixed at an active high level to allow it to count.

Select the net corresponding to the **Enable A** signal **ENA** (or whatever other name you have given it) within the *Pyxis Schematic* window, and click on **Forces/ICs** in the Tools palette, then select **Manager**. This will bring back the **Setup Simulation** dialogue box with **Forces** highlighted in the Simulation Panel. Note also that the net name you selected should appear in the **Selection from Schematic** list

A DC force of fixed voltage is required corresponding to Logic 1. To apply this, click on **DC** under **Source Type**, and set the **Magnitude (V/A)** to be 5 volts – this corresponds with the settings of the boundary converter elements mentioned above. Note that if Logic 0 were required, that would correspond to 0 V. Click **Add** to apply the force.

You can repeat this at any time during a simulation to change the force applied to **ENA**. In this case, highlight the specific signal in the table at the foot of the Setup Simulation dialogue with forces displayed, enter the new value in the **Magnitude (V/A)** box and click **Update**.

Finally, close the **Setup Simulation** dialogue.

18. Add a clock signal

The next type of signal to force is the periodic clock signal. Select the **CLK** signal in the schematic window. Open the **Forces/ICs** dialogue as previously.

In this case a periodic waveform is required. The corresponding **Source Type** is **PULSE**. You may note the strong resemblance between the range of sources (and their use) and packages like LTSpice.

With **PULSE** selected, the box will re-format to allow you to enter pulse parameters. The following recommendations are worth noting:

Initial Value (V/A) and **Final Value (V/A)**: one should be 5, the other 0.

Delay: the interval before the level switches from the intial to the pulsed value. A few ns is appropriate.

Rise Time (s) and **Fall Time (s):** 1ns is typical in the context of this project

Pulse Width: around half the clock period (see below)

Period: 10 to 50 ns, corresponding to frequencies in the range 20-100 MHz. The value chosen needs to be determined from the target design speed.

With these parameters set, click **Add**. Close the **Setup Simulation** dialogue box.

If you need to change any of these parameters later, use the procedure described in step 17, clicking on **Update** before closing the dialogue.

19. Add a reset signal

The easiest way to add a reset signal is to add it as a **PULSE** of the appropriate polarity, with a very long period and delay, and a short pulse width. Select the **RST** signal from the schematic window. **Note**: the period/delay need to be shorter than the simulation time set up in step 17. Use the **Forces/ICs** dialogue to edit the pulse parameters appropriately, and click **Apply** before closing the dialogue

Click **OK** to set the reset signal.

20. Select signals to plot

The results of your simulation are most usefully seen as graphical plots. Determine signals for plotting by selecting the signals that you wish to observe (e.g. **CLK**, **RST**, **ENA**, **COUNT**) in the schematic window. In the Tools palette, click on **Outputs. Note**: it is quite acceptable for the selection to include input signals. It can be valuable to see inputs and outputs on the same time axes.

In the **Setup Simulation** dialogue, all the signals you selected should be visible in the **Selection from Schematic** list. Check that **Analysis:** is set to **All**, **Type:** is set to **Voltage** and change **Task** to **Plot.** Check that the list of signals to be monitored is correctly reported at the foot of the dialogue box, expanding as necessary by clicking on the '+' signs. Click **Apply**, and close the dialogue box.

21. Run the simulation

You are ready to run the simulation. To launch it, click on the green arrow in the left toolbar. You will see a large number of diagnostics fly past in the **Transcript** area and the **Log**, and some messages may appear in the **Message Area**. However, nothing more serious than warnings should be seen; there should be no errors. In effect, *Pyxis* passes the simulation command via *Questa ADMS* to *vasim,* the VHDL simulator which runs outside *Pyxis* as a separate process, but returns the results so you can see them within *Pyxis* Schematic.

The simulation run may take several seconds to execute. It will terminate when the prescribed run time (set as 1000 ns in step 16) has elapsed.

22. View the waveforms and resume the simulation if necessary

As the simulation proceeds a database of waveforms is built up. To see these, click on the **Plot Results From Latest Run** icon in the left toolbar. This will bring up the *EZwave* panel, which after a few seconds will list the database corresponding to the cell you simulated.

Click on the '+' sign to open up the database, and click the '+' on the TRAN (transient) entry within. A lower pane should populate with the names of the signals you can see of the database With the **Ctrl** key pressed to allow multiple selections, click on the signals of interest. Then, with the right mouse button, chose the form of plot you require from the **Selected Waveforms** pull-down menu.

You can resume the simulation by clicking on the green arrow again. Repeating this will continue the simulation for another runtime length each time. By default, these results will overwrite the previous ones in the EZwave database. If you want to be able to see previous runs as well, use the *EZwave* **Edit > Options** command, and select the **Automatic Reload** feature. This allows you to opt to keep a specified number of earlier results, which will be accessed by tabs at the foot of the *EZwave* screen.

Do this as many times as you need to collect the results that you need. At this point, the plot inside *EZWave* should be populated with various timing graphs.

23. Measuring values and generating a plot

You can zoom in and out of the plot by using **View > Zoom In/Out** or by dragging your cursor across the time range of interest at the foot of the display. Use cursors on the plot to measure results. Add cursors by selecting **Cursor > Add** and dragging the cursor on the plot to points of interest.

Consider carefully what outputs you will need to generate to prove that your design is working within specifications and only generate the necessary ones. Once you have all the necessary information displayed on the plot, you can generate a JPEG or PNG image output for incorporation in your report.

Select **File > Export** and enter a suitable output file name (e.g. **count-eval-sim.jpg**). Be sure that you know where the output is being saved, as you will need to retrieve it from the file system for further processing using regular methods.

Developing the ring oscillator in VHDL

In the steps below, the entire procedure set out above is repeated to develop and simulate a ring oscillator, **ring_oscillator**, based on the concept of an array of 2-input NOR gates organised as detailed in the Project Guide. You will find a sample behavioural model that you can customise for such a gate in the text resource in **$MHOME/sb1_vhdl/sample**; a printed version is given at the end of this Lab Guide.

24. Consider the *nor2* model and review your design

Study the *nor2* behavioural model, which reflects the behaviour of a typical gate, and also incorporates a simple feature allowing programmable delay, which can be used to reflect the operation of a logic gate with typical load capacitances at its output. **Note**: **this model requires some updating** to match specific characteristics of the AMS NOR2 gate we shall use. It also contains at least one basic error, which you will be able to correct with ease provided you have studied and understood the listing.

Review your ring oscillator design and decide how many stages it needs to meet the timing specification detailed in the **Ring Oscillator Specification** section of the Project Guide. You will need to study the characteristics of typical NOR gates actually available, also in that section. Consider how to interconnect these stages.

25. Implementing the ring_oscillator design

Start implementation of your design by developing HDL code based on the *nor2* behavioural model to meet the specification. Look at the listing given at the end for the **ring_oscillator_vhdl** component. As supplied, this represents a model for a very simple 5-element oscillator. This will not meet the requirements for this design as it stands, and you will need to adapt it as necessary. It will need more stages.

When your own model is complete, save it and compile as for the **count4_vhdl** object (step 6). Correct any errors if necessary.

You will need to create a new schematic for the ring oscillator: **ring_oscillator.** Use the same methods as in steps 8 to 9 above to construct a symbol and a schematic named **ring_oscillator**.

Insert an instance of the **ring_oscillator** symbol, and incorporate appropriate input and output ports (step 10). Next, wire up the design until all necessary nets linking inputs and outputs have been created (step 11).

Check the schematic (step 12) before proceeding to name the nets suitably (step 13) – for example, **enb**, **out1** and **out2**. Check the schematic a further time, and save (step 14).

26. Preparing for simulation

Make preparations for simulation with *Questa ADMS*, along the lines of steps 15 to 16. Note that some of the setup activities were carried out previously, and do not need to be repeated.

27. Use *Questa ADMS* to verify the performance of the ring oscillator

Before running a simulation, it is necessary to set up the values of any external signals with the **Forces** menu. The methods used in steps 17-19 will apply. For the simulated ring oscillator to run, this requires that the signal **enb** be set to an appropriate value, using the tools in the **Forces/ICs** dialogue.

Use the advice in step 20 to select those signals you wish to monitor with *EZwave*.

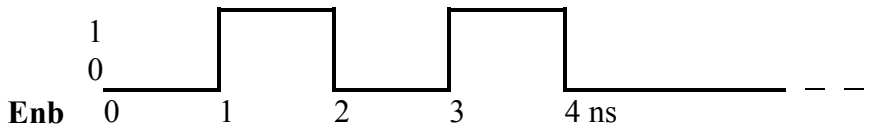
To run the simulation, follow the instructions in steps 21 and 22. You may wish to can alter the Stop Time in the **Analysis > TRAN** pane of the **Setup Simulation** dialogue.

You should now experiment with suitable `force` commands to characterise the behaviour of the ring oscillator.

If necessary, generate printouts of the *EZwave* waveforms for your records using the methods described in the *Getting Started* section.

28. Ring oscillator with pulsed Enable

Using the techniques illustrated in paragraph 29 above, investigate the expected behaviour of your own ring design (assuming the AMS NOR20 gate is installed – see specification in the section **Ring Oscillator Specification**) when a pulse of the form shown below is applied. **Note**: consider carefully what form and duration of initialisation signal needs to be applied to **ring_osc_enb** *prior* to the double pulse.



Examine the resultant output waveforms, paying particular attention to the period and the relative amount amount of time spent by each output in the logic **0** and logic **1** states.

Question Under these conditions - that is, with a short *double-pulse* applied to the **ring_osc_enb** input - does the ring oscillator appear to oscillate at its calculated operating frequency? If not, why is this so, and what is now the measured operating frequency? This represents the behaviour of the oscillator when the input is controlled by a mechanical switch, which may exhibit contact-bounce.

Hint: consider what you would measure if you were able to connect the output to a digital frequency counter, which simply counts the number of transitions passing in unit time.

29. Model the evaluation counter driven by your ring oscillator

Using *Design Architect-IC*, study the structure of the evaluation counter and your ring oscillator design and decide what connections to make so that the counter input is driven by one of the ring oscillator outputs.

Develop a top-level source (choose your own name) to reference the additional components.

- Establish port mappings so that one of the **ring_oscillator** outputs drives the clock input on the counter.
- Allow for the counter to be reset at will, and make its outputs available as external signals.

Plan your code carefully and use a hierarchical approach – the specimen code at the end of this guide should help. You should expect to spend some time away from the workstation preparing your design and the VHDL code to model it.

Objectives Use these resources to investigate the following:

Operation of the basic digital counter using a standard clock

- ♦ Demonstrate a full cycle of counting
- ♦ Show the operation of the manual Reset facility

Invoke simulations in the usual way and record the resultant waveforms.

Include the results of these investigations in your First Interim Report.

30. Investigate modified forms of the ring oscillator design (optional)

Investigate the effect on ring oscillator performance of using a NOR gate with delay for rising edges three times that for falling edges.

One way of doing this would be to revert to the VHDL source, modify the parameters T_{rise} and/or T_{fall} within it, and re-run the simulation. However, this would be fairly tedious.

A simpler and quicker method is to run Questa ADMS in a semi-standalone mode, rather than controlling all its features from *Pyxis*. To do this, click on the **Setup Environment** button in the command pallet at the right. Select **Simulator** in the **Environment Panel**, and in the **Options** section, check **Interactive**. Click **Apply**, and OK the box.

This time when you click the green **Run Simulator** button, once the simulation is complete, you will be presented with a different interactive environment to control its progress. This is *Questa ADMS's* own interface for HDL modelling, which we shall also meet later.

Although it would be possible to carry out much of the previous set of experiments in this environment, we shall confine ourselves here to a small change that cannot easily be carried out in *Pyxis*.

In the **Structures** pane at left, identify the architecture that corresponds to your **NOR2** gate, and highlight it by left-clicking once. In the **Objects** window you should now observe the named ports A, B and Y, as well as any constant parameters set up in the source code. These should include the delay parameters for rising and falling edges, `Trise` and `Tfall`. Select each of these in turn, and with the right mouse button, choose **Modify > Change Value** from the pull-down menu. Edit the numerical values as required and click **Change**.

Now, with the cursor in the **Transcript** pane at the foot, type the command:

`run 1000 ns` followed by **Enter.**

Although the simulator will issue a warning about modifying constants in this way, you should find that the output waveform now reflects the changed characteristics of the **NOR2** gate.

Question How do these new results compare with your expectations? If necessary, discuss your findings with a demonstrator.

Developing the programmable divider in VHDL

In this final section the team should select from, adapt and extend the available modules, to develop the divider architecture proposed for the target frequency synthesiser design. You will need at least one additional counter component, to achieve the required precision (number of channels); a comparator that can detect the state of certain of the counter outputs so the counter can be reset after a pre-determined number of counts, in effect becoming a programmable counter. Finally, you should add the necessary control structures, inputs and outputs, so this hypothetical model adequately emulates the target design. **Please note that while this will build on the 4-bit counter used in the VHDL familiarisation exercises above, it will be more complex.** The work may extend past the end of the scheduled session and into the final session of the week (Tuesday).

This part of the design should be shared between team members – the arrangement we suggest involves one member designing the multi-bit counter and any necessary control logic, and other one designing the comparator.

Once this has been achieved, work together to integrate these parts into the top-level system. When developing and testing the top-level design, you are recommended to proceed in stages, examining the behaviour of the new components separately before integrating them and closing the feedback loop, then finally adding the **ring_oscillator** module to add as a clock – see the test objectives below.

Objectives

To use the adapted models to investigate the behaviour of the counter in a programmable form. To develop a series of tests demonstrating the capability to achieve different division ratios. In a little more detail.

(a) Incorporation of a comparator with the counter:

- To use the comparator to detect a pre-determined counter state
- To demonstrate detection of at least three different states by using different combinations of **force** commands applied to the programming inputs

Edit the top-level source to extract a signal from the comparator that could be used to reset the counter to zero. Route this feedback signal to the counter Reset input. You will need to recompile the top-level source.

(b) Development of the complete programmable counter

- Demonstrate operation of the programmable counter counting modulo-N for at least three different N.
- Show how to produce a divided output with 1:1 duty cycle.

(c) Modify your design generates the required outputs reproducibly. Finally,

- Demonstrate that the ring oscillator is capable of driving the programmable counter.
- Show that the design counts correctly when provided with the ring oscillator input and appropriate programming inputs. Produce a timing diagram showing inputs and outputs to demonstrate correct operation for at least three different programmed inputs.

Include the results of these investigations in your First Interim Report. You are recommended to keep a copy – you will need it for Lab Session 4.

In this section, to save some time, and to give you some experience of using the Questa ADMS package in a stand-alone form, we shall dispense with the creation of new schematics for every stage of development. Planning and HDL code development can be done away from the workstation if necessary; source-code entry is done using a standard text-editor. You may be able to re-use some elements of your earlier work, and the VHDL sources at the end of this section should still be of use. Preparation for simulation itself is done at the Linux command line, with new modules being compiled using *vcom*, and any errors corrected before proceeding to simulation, which is done in a GUI using the Mentor *vasim* simulator and the *EZWave* display utility.

The final design will incorporate the ring oscillator already considered as clock input to the counter. This will represent the voltage-controlled oscillator (VCO) used in the synthesiser.

Working together, adapt the **top level** source to incorporate this feature and any others you consider important. You should plan your code carefully and use a hierarchical approach – the specimen code at the end of this guide should help. You should expect to spend some time away from the workstation preparing your design and the VHDL code required to model it. Incorporate additional models for combinational and sequential logic elements, as required. You can see listings of these at the end of this section.

The broad procedure is set out below. Study and adapt the appropriate VHDL elements and resimulate as necessary until you are happy with the results.

31. Preparing for standalone HDL simulation

We recommend you carry out this part of the work outside the project structures you have set up for work with Mentor *Pyxis*, since *Pyxis* is not involved in any of the operations that follow.

Start with a terminal window logged in to the Mentor server, nd run the start-up script **`pyxis`** in the normal way. However, instead of starting up the *Pyxis* Project Manager, change directory to `$MHOME/cbt_hdl`, which has been created for this work. Give the commands:

```
cd $MHOME/cbt_hdl          # The directory for new HDL work
vlib sb1_synth             # Create work directory for vcom/vasim
vmap sb1_synth             # Register directory with vcom/vasim
vmap WORK sb1_synth        # Register the WORK directory
```

You are now ready to create and compile the required architecture VHDL files. Although files can be named as you choose, it is best to use a standard convention. The standard naming convention for the VHDL architecture source for a design RING_OSC would be:

`RING_OSC.arch.vhd`

To save confusion later, you should adopt this convention as you create the necessary files.

As you develop the code for each part of your design, you should compile it to check there are no errors. Note that for VHDL, compilation is a bottom–up

process. The compiler will flag an error if any references to lower-level parts of the design cannot be satisfied at time of compilation. You should therefore start with (to take a simple example) the NOR2 gates used in the ring oscillator, followed by the ring oscillator itself, and finally the top-level structure that invokes the ring oscillator.

For compilation, make sure all the necessary source files are placed in the **`cbt_hdl`** directory, and that your terminal screen is open on that directory, and the Mentor *Pyxis* start-up script has run. For the source file shown above, the command to compile is:

```
vcom RING_OSC.arch.vhd
```

Any errors found will be flagged in the terminal window, and these must be corrected in turn to get a clean compile. This will probably require a number of attempts. In practice it is probably best to correct only the first error flagged before re-compiling – a single missing semi-colon can generate dozens of errors in subsequent lines that do not in fact need correction. If you are fortunate and there are no errors, the compiled version will appear in the **WORK** directory **`sb1_synth`**.

When all module files have gone cleanly through compilation, you are ready to simulate with *vasim*.

32. Use *Vasim* to verify the performance of the complete design

Start the simulator by entering the **vasim** command:

```
vasim
```

This launches the *Questa ADMS* environment, which is fully capable of handling mixed-signal simulations, though we can confine ourselves to the features supporting VHDL simulation. The **Load Design** dialogue should appear; navigate to the top-level element of your model. Expand it by clicking on the associated '+' symbol, and note that if you provided more than one model, this provides a means of selecting which one should be used.

The lower pane allows selection of a command file – a list of commands in the Tcl scripting language – which could be used to configure and control the simulation. You may want to consider developing such a script, but for the moment, ignore this pane and OK the dialogue.

Vasim will configure the full set of simulation windows, comprising **Architecture, Objects, Processes, Locals, Transcript,** and **Waves**.

The following are some hints and tips for setting up and running simulations.

Before running the simulation, select signals for display in the **Waves** panel. Hold down **Ctrl** to allow multiple selections, then click each one.

Click in the **Transcript** window to enter commands – for example, `force`, `run`. Type the commands and terminate with **Enter.**

To apply a force, the simplest way is to type in commands – e.g. `force enb 1`, though you can also select the signal to which you wish to apply a force, and with the cursor in the **Objects** window, click **(RMB)** > **Modify** > **Apply Clock**. In this pulldown can also be found interactive commands to add or remove forces.

In *vasim*, you can specify the time unit for delays in all simulator commands that have time arguments. For example:

```
force clk 1 50 ns, 1 100 ns -repeat 1 us
run 2 ms
```

Note that the time units used in a command need not be the same.

Unless you specify otherwise as in the examples above, simulation time is always expressed using the resolution units that are specified by the **UserTimeUnit** variable in the *vsim* section of the `modelsim.ini` file, which you should find in the directory **cbt_hdl**.

♦ ♦ ♦

Appendix – VHDL Source Listings

The following section contains VHDL source listings for reference. All models invoke the **IEEE** library which contains definitions of the signals that are accepted by the Modelsim simulator. A number of alternative libraries are available, including some specialist ones contributed by device manufacturers.

The **IEEE** library package contains a basic 4-state data type called **std_logic**; three of these states (**0**, **1**, **Z** - for high impedance) can be used for describing designs: the fourth state **X** stands for 'unknown'. Included also are definitions of all relational operations based on these states. Other, more complex data types are defined, along with supporting resolution, conversion, and operator functions, but these are not required for this project.

More details of the **IEEE** library are available on the web at:

```
http://en.wikibooks.org/wiki/Programmable_Logic/VHDL_Data_Types
```

VHDL source listings for reference

```
-- 2 INPUT NOR GATE

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY nor2_vhdl IS
      PORT(A, B : IN std_logic; Y : OUT std_logic);
END nor2_vhdl;

ARCHITECTURE behav OF nor2_vhdl IS
 CONSTANT Trise: time:= 10 ns; -- Typical delay, 0-1 transition
 CONSTANT Tfall: time:= 12 ns; -- Typical delay, 1-0 transition

-- Trise and Tfall values are indicative.
-- Their values must be changed to correspond to the actual design.
-- All references to Trise/Tfall below must be carefully checked!!

BEGIN
 nor_inputs : PROCESS (A,B)
 BEGIN
  IF (A OR B) = '1' THEN
   Y <= '0' AFTER Trise;     -- Check this statement carefully
  ELSIF (A AND B) = '0' THEN
   Y <= '1' AFTER Trise;
  END IF;
 END PROCESS nor_inputs;
END behav;

-- TEMPLATE FOR SIMPLIFIED RING OSCILLATOR (5 STAGES)

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY ring_oscillator_vhdl IS
      PORT (            ENB   : IN  std_logic ;
                        OUT1  : OUT std_logic ;
                        OUT2  : OUT std_logic   );
END ring_oscillator_vhdl ;

ARCHITECTURE struct OF ring_oscillator_vhdl IS
      COMPONENT nor2v
            PORT (A, B : IN std_logic; Y : OUT std_logic);
      END COMPONENT;

      SIGNAL nor_out : std_logic_vector(0 TO 4);

      FOR u1, u2, u3, u4, u5: nor2v USE ENTITY WORK.nor2_vhdl(behav);
      BEGIN
            u1:nor2v PORT MAP(ENB,nor_out(4),nor_out(0));
            u2:nor2v PORT MAP(nor_out(0),nor_out(0),nor_out(1));
            u3:nor2v PORT MAP(nor_out(1),nor_out(1),nor_out(2));
            u4:nor2v PORT MAP(nor_out(2),nor_out(2),nor_out(3));
            u5:nor2v PORT MAP(nor_out(3),nor_out(3),nor_out(4));

            OUT1 <= nor_out(4);
            OUT2 <= nor_out(1);
END struct;
```

```
-- INITIAL TOP-LEVEL DESIGN INCLUDING ONLY RING OSCILLATOR

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY ring_vhdl IS
      PORT( ring_osc_enb            : IN std_logic;
            OUT1, OUT2              : OUT std_logic;
            );
END ring_vhdl;

ARCHITECTURE struct OF ring_vhdl IS
      COMPONENT OSCILLATOR
            PORT (ENB: IN std_logic; OUT1, OUT2 : OUT std_logic);
      END COMPONENT;

      signal osc1, osc2             : std_logic;

      FOR u1: OSCILLATOR USE ENTITY
            WORK.ring_oscillator_vhdl(struct);

      BEGIN

            u1:OSCILLATOR PORT MAP(ring_osc_enb,osc1,osc2);

            OUT1 <= osc1;
            OUT2 <= osc2;

END struct;


-- 4 BIT SHIFT REGISTER WITH PRESET

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY shift_vhdl IS
PORT( data_in, clock, preset : IN std_logic;
                        shift_out  : OUT std_logic_vector(0 TO 3));
END shift_vhdl;

ARCHITECTURE behav OF shift_vhdl IS
      CONSTANT Tdelay    : time := 10 ns; -- Typical delay
BEGIN

check_clock : PROCESS(clock, preset)
      VARIABLE internal_out : std_logic_vector(0 TO 3);
BEGIN
      IF (preset = '1') THEN
            internal_out := "1111";
      ELSIF ( clock'last_value = '0' and clock = '1' ) THEN
            internal_out(3) := internal_out(2);
            internal_out(2) := internal_out(1);
            internal_out(1) := internal_out(0);
            internal_out(0) := data_in;
      END IF;
      shift_out <= internal_out AFTER Tdelay;
      END PROCESS check_clock;
END behav;
```

```
-- 4 BIT COUNTER WITH RESET

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY count4_vhdl IS
      PORT(clock, reset : IN std_logic;
      data_out          : OUT std_logic_vector(0 TO 3));
END count4_vhdl;

ARCHITECTURE behav OF count4_vhdl IS
      CONSTANT Tdelay   : time := 10 ns; -- Typical delay

BEGIN

check_clock : PROCESS(clock, reset)
VARIABLE count  : std_logic_vector(0 TO 3) := "0000";
BEGIN

IF ( reset = '0') THEN
      count := "0000";
      ELSIF ( clock'last_value = '0' and clock = '1') THEN
            IF (count = "1111") THEN
                  count := "0000";
            ELSE
                  count :=  count + "0001";
            END IF;
END IF;
      data_out <= count AFTER Tdelay;
END PROCESS check_clock;
END behav;


-- 2 INPUT XOR COMPONENT

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY xor2_vhdl IS
      PORT(A, B : IN std_logic; Y : OUT std_logic);
END xor2_vhdl;

ARCHITECTURE behav OF xor2_vhdl IS
      CONSTANT Trise: time:= 20 ns; -- Typical delay, 0-1 transition
      CONSTANT Tfall: time:= 20 ns; -- Typical delay, 1-0 transition

-- Trise and Tfall values are indicative and may not apply to the
-- Mietec process used in this project

BEGIN
  xor_inputs : PROCESS (A,B)

  BEGIN
   IF (A XOR B) = '0' THEN
      Y <= '0' AFTER Tfall;
   ELSIF (A XOR B) = '1' THEN
      Y <= '1' AFTER Trise;
   END IF;
  END  PROCESS xor_inputs;
END behav;
```

```
-- OTHER COMPONENTS

-- 2 INPUT AND GATE

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY and2_vhdl IS
      PORT(A, B : IN std_logic; Y : OUT std_logic);
END and2_vhdl;

ARCHITECTURE behav OF and2_vhdl IS
 CONSTANT Trise: time:= 8 ns; -- Typical delay for 0-1 transition
 CONSTANT Tfall: time:= 6 ns; -- Typical delay for 1-0 transition

-- Trise and Tfall values are indicative only
-- Their values must be adjusted to correspond to the actual design.

BEGIN
 PROCESS (A,B)
 BEGIN
  IF (A AND B) = '1' THEN
   Y <= '1' AFTER Trise;
  ELSIF (A AND B) = '0' THEN
   Y <= '0' AFTER Tfall;
  END IF;
 END PROCESS;
END behav;


-- 3 INPUT AND GATE

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY and3_vhdl IS
      PORT(A, B, C : IN std_logic; Y : OUT std_logic);
END and3_vhdl;

ARCHITECTURE behav OF and3_vhdl IS
 CONSTANT Trise: time:= 9 ns; -- Typical delay for 0-1 transition
 CONSTANT Tfall: time:= 9 ns; -- Typical delay for 1-0 transition

-- Trise and Tfall values are indicative only
-- Their values must be adjusted to correspond to the actual design.

BEGIN
 PROCESS (A,B,C)
 BEGIN
  IF (A AND B AND C) = '1' THEN
   Y <= '1' AFTER Trise;
  ELSIF (A AND B AND C) = '0' THEN
   Y <= '0' AFTER Tfall;
  END IF;
 END PROCESS;
END behav;
```

```
-- SIMPLE INVERTER

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY inverter_vhdl IS
      PORT(A : IN std_logic; Y : OUT std_logic);
END inverter_vhdl;

ARCHITECTURE behav OF inverter_vhdl IS
 CONSTANT Trise: time:= 5 ns; -- Typical delay for 0-1 transition
 CONSTANT Tfall: time:= 5 ns; -- Typical delay for 1-0 transition

-- Trise and Tfall values are indicative only.
-- Their values must be adjusted to correspond to the actual design.

BEGIN
 PROCESS (A)
 BEGIN
  IF (A) = '1' THEN
   Y <= '0' AFTER Tfall;
  ELSIF (A) = '0' THEN
   Y <= '1' AFTER Trise;
  END IF;
 END PROCESS;
END behav;


-- 2 INPUT INCLUSIVE OR GATE

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY or2_vhdl IS
PORT(A, B : IN std_logic; Y : OUT std_logic);
END or2_vhdl;

ARCHITECTURE behav OF or2_vhdl IS
  CONSTANT Trise: time:= 6 ns; -- Typical delay for 0-1 transition
  CONSTANT Tfall: time:= 6 ns; -- Typical delay for 1-0 transition

-- Trise and Tfall values are indicative only

BEGIN
  PROCESS (A,B)

  BEGIN
   IF (A OR B) = '0' THEN
      Y <= '0' AFTER Tfall;
   ELSIF (A OR B) = '1' THEN
      Y <= '1' AFTER Trise;
   END IF;
  END  PROCESS;
END behav;
```

```
-- D-TYPE BISTABLE (WITH RESET)


LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY d_ff_rl_vhdl IS
      PORT(clock, reset, data_in    : IN std_logic;
            data_out, data_out_bar  : OUT std_logic);
END d_ff_rl_vhdl;

ARCHITECTURE behav OF d_ff_rl_vhdl IS

 CONSTANT Tdelay    : time := 10 ns; -- Typical delay

BEGIN

      PROCESS(clock, reset)

      BEGIN
            IF ( reset = '0') THEN
                  data_out <= '0';
                  data_out_bar <= '1';
            ELSIF ( clock'last_value = '0' and clock = '1') THEN
                  data_out <= data_in AFTER Tdelay;
                  data_out_bar <= NOT data_in AFTER Tdelay;
            END IF;

      END PROCESS;
END behav;




-- COMPLETE COUNTER AND COMPARATOR DESIGNS ARE LEFT
-- TO INDIVIDUAL DISCRETION.  TEMPLATE DESIGNS MAY
-- BE BASED ON THE METHODS OUTLINED ABOVE
-- AND THE OUTLINE SPECIFICATION
-- PROVIDED BELOW
```

```
-- BASIC VHDL TEMPLATEs FOR SIMPLE COUNTER a
-- Note that behavioural and structural models are presented.

-- BEHAVIOURAL MODEL

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY ClockDivider IS

generic(Modulus: in Positive range 2 to Integer'High);
      PORT( ClkIn: in std_logic;
            Reset: in std_logic;
            ClkOut: out std_logic);
END ClockDivider;

ARCHITECTURE behav OF ClockDivider IS
begin
process (ClkIn, Reset)
      variable Count: Natural range 0 to Modulus-1;
      begin
      if Reset = '1' then
            Count := 0;
            ClkOut <= '0';
      elsif ClkIn = '1' and ClkIn'event then
            if Count = Modulus-1 then
                  Count := 0;
            else
                  Count := Count + 1;
            end if;

            if Count >= Modulus/2 then
                  ClkOut <= '0';
            else
                  ClkOut <= '1';
            end if;
      end if;
end process;
end behav;

-- STRUCTURAL model (template only)

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY COUNTER_VHDL IS
PORT
(INPUT : in std_logic;
 OUTPUT  : out std_logic_vector(0 to XX)
);

END COUNTER_VHDL;

ARCHITECTURE arch OF COUNTER_HDL IS
.

BEGIN
.
.
.
END arch;
```

```
-- STRUCTURAL TEMPLATE FOR TOP-LEVEL DESIGN
-- NOTE: This template is indicative only and will
--    need considerable adaptation for your purposes
-- In particular, the size of state_vectors, counters & comparators
--    will depend on the level of design chosen
-- It will need further extension for a pulse-swallow design

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.All;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY top_vhdl IS
      PORT( ring_osc_enb, reset     : IN std_logic;
            OUT1, OUT2, DIV_out     : OUT std_logic;
            Compare_inputs          : IN std_logic_vector(0 TO ??)
            );
END top_vhdl;

ARCHITECTURE struct OF top_vhdl IS
      COMPONENT OSCILLATOR
            PORT (ENB: IN std_logic; OUT1, OUT2 : OUT std_logic);
      END COMPONENT;

      COMPONENT COUNTER
            PORT (clock, reset: IN std_logic;
                  data_out    : OUT std_logic_vector(0 TO ??));
      END COMPONENT;

      COMPONENT COMPARATOR
            PORT (compare_A   : IN std_logic_vector(0 TO ??);
                  compare_B   : IN std_logic_vector(0 to ??);
                  comp_out    : OUT std_logic);
      END COMPONENT;

      COMPONENT CONTROL_LOGIC
            PORT(comp_in, ext_reset : IN std_logic;
                  reset             : OUT std_logic);
      END COMPONENT;

      signal osc1, osc2             : std_logic;
      signal divider_out            : std_logic;

      FOR u1: OSCILLATOR USE ENTITY
            WORK.ring_oscillator_vhdl(struct);
      FOR u2: COUNTER USE ENTITY
            WORK.count??_vhdl(struct);
      FOR u3: COMPARATOR USE ENTITY
            WORK.comparator_vhdl(struct);
      FOR u4: CONTROL_LOGIC USE ENTITY
            WORK.control_logic_vhdl(struct);

      BEGIN

            u1:OSCILLATOR PORT MAP([to be completed] osc1, osc2);
            u2:COUNTER PORT MAP([to be completed]);
            u3:COMPARATOR PORT MAP([to be completed]);
            u4:CONTROL_LOGIC([to be completed]);

            OUT1 <= osc1;
            OUT2 <= osc2;
            divider_outputs <= DIV_out;

END struct;
```

Standard Project on VLSI Design SB1

Laboratory Guide 3 – Synthesis with Leonardo Spectrum

This section of the project illustrates one approach to design synthesis. It uses as input data a design specified in a hardware description language, and yields as its output a *netlist* defining how suitable cells from an appropriate library of electronics parts may be interconnected to provide a physical implementation.

This is an attractive, relatively low-effort means of developing electronic circuits. It is expected to be intrinsically free from error, since the input data can be trusted to the extent that it has passed through simulation to the satisfaction of the designer. No manual post-processing should be required, so the scope for errors committed inadvertently by the designer should be small. However, the effectiveness of the method depends directly on the extent to which the simulation models used reflect the behaviour of the library, and the effectiveness of the synthesis algorithm in matching models to library functions. For these reasons, direct synthesis is normally confined to digital design, in which the models and library cells can be characterised very accurately. It is not normally a satisfactory approach to implementation of analogue cells, where the specifics of layout, tolerances, and other factors may have major impact on performance.

Mentor Graphics' *Leonardo Spectrum* is a multi-target synthesis tool. It can accept data from design sources in a variety of hardware design languages, optimise the design for either timing or area constraints, and map the design in terms of a specific technology. Quite remarkably, even though the input data may be expressed in a variety of forms and models, behavioural or structural, *Leonardo* is capable of synthesising correct logic implementations under most circumstances.

In this section, you will use direct synthesis from VHDL to develop a cell-based design for the two key digital elements of the synthesiser design: the counter and the comparator. The instructions given below are generic in nature: the two team members should decide which of the two elements each will develop, and should work on them simultaneously at separate workstations, adapting the instructions given according to whether it is the counter or the comparator being designed. Later, you will work together to amalgamate your designs into one project, combining them with other library cells and blocks created in other ways (for example, from schematics).

1. Start Leonardo Spectrum

You should only attempt this after you have successfully designed and completed all functional simulation, and have the required **.vhd** VHDL sources readily available, free from errors.. You should only need to run synthesis for the top level of the chosen part of your design. There is no need to synthesise every sub-level individually.

Log in to the Mentor server in the usual way, and run the *Pyxis* start-up script from a terminal window. There is no need to start *dmgr_ic* unless you wish to for some other reason. Since *Leonardo* is not integrated into the *Pyxis* design flow, we need to use it in a stand-alone mode.

In the terminal window, give the following command to switch to a suitable working directory:

```
cd $MHOME/leonardo
```

The **leonardo** directory was created outside the **$CBT_WD** directory tree to reflect the independence of these approaches.

Now start up *Leonardo* by giving the following command:

```
leonardo
```

Leonardo should start up, and after displaying an introductory splash screen you should see its graphical user interface. This is discernibly different from the *Pyxis* tools, in consequence of its different origins. There are three User Levels available with *Leonardo;* we use Level 3, which is the most feature-rich. *Leonardo* is underpinned by a scripting language, Tcl, in which all the necessary commands for this session could be given at the keyboard. For convenience, however, and to get a feel for the range of options, we shall use the GUI where possible. A small number of commands must be given as Tcl commands, as they are not supported by the GUI. The Tcl **Command Entry** box is on the right in the lower part of the window. An **Information** pane, in which transcripts and reports are presented, lies in the upper part of the window.

At startup, the fonts used in the **Information** and **Command** panes are rather small. You can use the **(menu bar) > Tools > Configure Document's Colours** to adjust the font size, e.g. to 12-point. This needs to be done for both of the panes. First place the cursor in the desired pane, set the font size, then move the cursor to the other pane, and repeat.

By default, Leonardo starts up in **Flow Tabs** mode, with a set of tabbed dialogues to control the synthesis process in the left-most pane. Setting up is basically a question of selecting each tab in turn and making adjustments as necessary in each of the dialogues that results. Various **Advanced Settings are**s also available via tabs at the foot of the screen, giving very fine control of the synthesis process, but the results obtained from the standard dialogues should be sufficient for this project.

2. Load the custom CUED rules

It is first necessary to load a set of locally-developed rules, a set of Tcl commands. These are required later in the process to rename the synthesised cells to prevent clashes with the original functional HDL code. To load the rules, place the cursor in the **Command Entry** pane and type in the following command:

```
source /tools/kits/mentor/ams/C35/cuedlib/synlib/cued.rules
```

followed by **Enter**. Unless there is some sort of error, Leonardo will execute the command silently, and will read in a ruleset named **CUED**. To check that it has done so, type in the following command:

```
report_rename_rules
```

followed by **Enter**, and check that **CUED** is included in the list of rules presented in the **Information** pane.

The remainder of the process now largely consists of navigating the **Flow Tabs**.

3. Select a target technology

Synthesis is target-technology-dependent. Therefore, you will need to select the correct technology library to use. There are a number of built-in libraries for Leonardo Spectrum, most of which correspond to technologies based on FPGAs. We require a technology suited for use in Application-Specific IC (ASIC) design.

A custom synthesis library corresponding to the AMS C35 cell library has been prepared. To load this, first select the **Technology** tab, and in the dialogue that results, expand the **ASIC** item by clicking on the adjacent '+' symbol. This should reveal an item: **AMS**, which, when expanded in the same way, will expose a number of libraries including **c35_CORELIB**. Select this and click the **Load Library** button to load it as your default library. It is theoretically possible to load multiple libraries, but this is not advisable.

4. Set design constraints and target speed

Synthesis is a *constraint-driven* process. The basic constraint that needs to be set is the target speed. This should be set to 100 MHz, (or whatever speed corresponds to the project specifications). There are other constraints that can be set depending on the target technology.

To set these, click on the **Constraints** tab and check that the following is set up:

Specify Clock Frequency = 100MHz

There is no need to specify delays in the panel below, since for simple setups these are implied by the chosen clock frequency. Click **Apply.**

If you are so inclined, you may wish to investigate the different results obtained with different constraints. At the very least, you should explore the synthesis results with a different speed constraint. You can get access to even more constraint settings by using the **advanced tabs** at the foot of the panel.

5. Add source HDL files, check the working directory, synthesise

Switch to the **Input** tab by clicking it.

Now manually add in all the HDL design files needed (only one may be required). Use the browser to the right of the **Open Files** text box to navigate to the directories where you prepared the HDL files, and, in turn, select all those that are required for synthesis.

You may wish to check the working directory as well. The working directory is where Leonardo will place all the output files generated. By default this will be the directory in which the *leonardo* command was given in step 1, which should be **$MHOME/leonardo**. At the very least, you should note down the location of the default working directory. If *Leonardo* reports that the directory is not correctly set, amend it by using the browser to the right of the Set Working Directory text box.

No other changes should be required, but check the following are set:

Encoding Style – Auto
Resource sharing – Checked
Run Elaborate – Checked
Run Pre-optimization – Checked

When the setup is complete, click **READ**. *Leonardo* will read in the **.vhd** files and proceed with synthesis. Watch the information pane for messages.

If all goes well, the synthesis process will take a few minutes. If there are errors, you will need to fix your HDL code before proceeding. At this point, you may discover the fact that some HDL code may not be synthesisable and will generate unusable output results.

6. Set up optimisation

Leonardo can be programmed to optimise the design in a number of ways. We shall optimise for minimum area. To check the settings for this, click on the **Optimization** tab, and ensure that the following settings are selected:

Optimize for: Area
Hierarchy: Auto
Run timing optimization – Checked

Do not select any other options; click the **OPTIMIZE** button, and watch the **Information** panel for messages

7. Specify the output netlist file.

Although *Leonardo* can generate a number of different formats, only a **Verilog** netlist can subsequently be imported by *Pyxis* Schematic.

Assuming you have got this far without errors or other problems, you now need to specify an output netlist file in which to save the results of the synthesis. Click the **Output** tab to do this. Verify the following settings:

Format: Verilog (note default filename, based on the corresponding input file)
Write vendor constraints file – Checked
Pre-process net list – Checked
Down to technology cells – Selected

Make sure that your netlist file ends with a **.v** extension. If you wish to give it a more useful name, you can do so – for example, something like **COUNT*nn*.arch.v** or similar, where *nn* represents the number of counter stages, for example, 16. Maintain this notation in the instructions that follow.

This file will be placed in the *Leonardo* working directory.

8. Rename the Cells

The synthesised cells need to be renamed to prevent clashes with the original functional HDL code. If you have kept their original names, which end with VHDL, this can be done automatically. In the Tcl command window, type in the following commands:

```
apply_rename_rules -ruleset CUED
auto_write <verilog netlist filename> (should end with a .v extension)
auto_write <vhdl filename> (should end with a .vhd extension)
auto_write <sdf filename> (ends with a .sdf extension)
```

The first command executes our custom renaming rules which apply to the references within the netlist, though not to the name of filenames used for the netlists in the following commands. For these you should choose suitable names. The second

command will re-write the previously generated Verilog netlist. The third command writes out a VHDL equivalent of the netlist for the synthesised design, almost ready for running through the VHDL simulation process. The fourth command generates an SDF file that contains timing information for simulation. It is essential that these steps be performed correctly to differentiate the pre- and post-synthesis models during simulation.

9. Viewing synthesised output

You can view your synthesised design both in generic technology and in target technology. This will be useful when trying to debug parts of your design. You may find that some parts of your HDL code may not synthesise correctly.

Select **Tools > View RTL Schematic** to view the generic technology schematic. This shows the HDL design in functional blocks. This is useful for investigating how the HDL code is translated into functional hardware blocks. You can also verify functionality at this stage.

Select **Tools > View Gate Level Schematic** to view the standard cell schematic. This shows how each functional block is then translated into available logic primitives. This can be useful for investigating how synthesis constraints affect the translation of functional blocks into resultant logic.

In order to understand what each standard cell does, please refer to the accompanying design kit documentation in CamTools, Project Documents page. Scroll down to document *AMS 0.35µm CMOS Digital Standard Cell Databook*.

10. Generate area/timing reports

Once you are satisfied with the results, you can generate area and timing reports.

Click on the **Report** tab. You will see two sub-tabs at the bottom: **Report Area** and **Report Delay**, which will generate area and timing reports respectively. Enter a file name for each of these in turn in the **Report Filename** box, or use the browser if needed; click on **Report Area** and **Report Delay** to generate the reports.

If there are any problems meeting the specifications of the design, these reports will prove useful in identifying the critical path of the design. You can then make changes to your HDL code to correct any problems.

Post Synthesis Simulation

11. Rename the top level Netlist design

Although in step 8 we renamed the cells to end with **_SYN** rather than **_VHDL**, in order to guard against clashes between them and the original VHDL code, it is still possible that some pre- and post-synthesis models may have the same names. This could create confusion in any attempt to simulate the post-synthesis models in *Questa ADMS*. You should be aware of this when importing or copying these models to other libraries, and rename them if necessary.

12. Import the Verilog Net list into *Pyxis* Schematics

If it has not already been started, start up *Pyxis* Schematic. Then, choose a library inside your project in which to save the schematic and symbol to be created by this

step. If necessary create a new one, as follows: revert to *Pyxis* Project Manager for a moment, and with your Project open and highlighted, give the command: **File > New > Library**, enter the chosen name, and OK the dialogue.

Return to *Pyxis* Schematics. Now select **File > Import Verilog** to bring up the **Import Verilog** dialogue.

In the **Netlist Files** pane, use the Navigator to select the Verilog **.v** file you created in step 8. Check carefully that this has registered correctly in the text box.

In the **Mapping Files** pane, enter the prefix: **`$AMS_MAP`**.

In the **Output Directory** pane, navigate to and select the library chosen or created earlier in this step.

In the lower part of the dialogue, where you see:

For existing components in the output directory, the option: **Do not replace** should be checked. No other buttons should be checked.

Click **OK** to begin the process. This may take several minutes as the design is read in and mapped onto the **C35_CORELIB** library. It should complete without any errors and create new schematic and symbol views, which will be visible within the **[design]_SYN** cell. You should view the schematic and symbol by double-clicking on them. Make a visual check that the synthesised design is broadly correct in terms of the expected numbers of inputs and outputs, bistables for a counter, etc.

Important: neither the symbol nor the schematic will have undergone formal checking. You should do this using **File > Check Schematic** or **File > Check Symbol**, as appropriate. If there are errors relating to power and ground nets, you may wish to make a small edit to address these. Place two short pieces of interconnect in the cell, unconnected to anything else. Set the **Net Names** to **vdd** and **gnd** respectively. Re-check and save when satisfied.

13. Prepare the synthesised VHDL model for use in Questa-ADMS

The visual check carried out in step 12 is not sufficient, of course, and synthesis packages cannot be assumed to produce a guaranteed-correct result. We need to simulate the post-synthesis model and compare it for correctness with the pre-synthesis model.

For the purposes of simulation, most of the **CORELIB** cells are linked to transistor-level schematics for which a very detailed simulation using *Eldo* can be run. We shall explore this in **Lab 4**. In addition, VHDL models have been compiled for these cells, and they include basic timing information. As a result, it is possible to take the structural VHDL output that can be generated by *Leonardo*, and use it to set up a simulation of the synthesised design. This is the course we shall pursue for the moment.

The VHDL file prepared in step 8 will have numerous references to **CORELIB** cells, but no **library** statement to allow it to access the models for these. We must remedy this before it will be possible to simulate the synthesised model. There are two parts to this: firstly, inserting a **library** reference in the model file; secondly, declaring the path to the library itself.

Using the Gnome Terminal (make sure it is a terminal window on **clae0X** and not one on the Teaching System) and navigate to the **leonardo** directory where the output from *leonardo* was created. Open the VHDL file in **gedit** or your preferred text Editor.

Note: you can alternatively navigate to the directory using *Pyxis* Project Manager, noting that it lies outside your project/s in the **$MHOME** directory, then use Mentor's *Notepad* app.

You should see the following text near the top (line numbers are typical):

```
10: --      LeonardoSpectrum Level 3, 2014a.12
11: --
12:
13: library IEEE;
14: use IEEE.STD_LOGIC_1164.all;
```

At line 12, insert the following (do not type in the line numbers):

```
12:
13: library C35_CORELIB;
14: use C35_CORELIB.vcomponents.all;
15: use C35_CORELIB.vtables.all;
16:
```

Save the file.

14. Create a simulation library and import the synthesised VHDL model

Start by creating a library within your project to hold the synthesised VHDL model. Use the procedure of step 7 in Lab Guide 1. The library name may be of your own choosing, but something along the lines of **counter_syn** or **comparator_syn** might be appropriate. With your project highlighted, open the language interface with **File > Open > Language Interface**.

First set up a mapping to the directory that actually contains the HDL models for C35_CORELIB. With the language interface window active, give the command: **Edit > Manage INI mappings …** . When the dialogue box opens, note that it should contain a few entries already, including one for your projects and one for a ***vasim*** WORK directory. If there is already an entry for **C35_CORELIB**, omit the instruction below, and import the file as in the next paragraph**.**

Click on **Add Mappings**. In the resulting **Compiled Library Navigator**, navigate to the following path; there is no short-cut browser for this part of the set up, so you need to type into the **Look in:** text box:

/tools/kits/mentor/ams/C35/cuedlib/modelsim/C35_CORELIB

A list of library directories starting **add21** should appear; do not select any of these but just click **OK**. A new entry **C35_CORELIB** should appear, with the path you just keyed in.

Now import the HDL file, using the command: **File > Import HDL**. Use the basic procedure of step 5 in Lab Guide 2. In the **Import HDL** dialogue that appears, navigate to the **.vhd** file in the **Leonardo** folder you edited in step 13, enter a cell name into which to import, and click **OK**.

This should compile the model, create a symbol and register the compiled model to the symbol, within the chosen cell. As the model was generated by *Leonardo* it should be free from errors, barring accidents in step 13!

At this stage you may also want to think of a way to compare the results of pre- and post-synthesis models. Read the following paragraphs before making a start.

One possibility is to consider incorporating both pre- and post-synthesis models within the same testbench schematic. You can do this by editing your earlier testbench schematic (used to test behavioural models) and adding the synthesised model in place, or creating a new one with both models. You can in principle run both the simulation models side-by-side. Take care to avoid naming clashes, as already mentioned.

There is of course nothing wrong with simulating only the post-synthesis model and comparing the output plots manually.

15. Import the synthesised VHDL model and create a test schematic

Using a similar method to that described in **Lab Guide 2**, steps 8 to 15, within the new library used in step 13 above, create a testbench schematic for this post-synthesis model.

Instantiate the symbol for the synthesised VHDL model and place and name pins and wires as required. **Note**: you may need to learn about placing a bus and/or a bus-ripper if your design has many inputs or outputs organised as a bus.

16. Prepare and run the simulation

Using the approach described in **Lab Guide 2**, steps 16 to 21 and beyond, set up a suitable simulation run and gather data in EZwave to demonstrate that your design meets specification.

Note that it would also be possible to use the Verilog **.v** file produced by *Leonardo* to run the simulation. *Pyxis* and *Questa ADMS* are equally at home with either. However, to spare you the ennuie of having to learn yet another HDL, we elected to use this slightly redundant approach.

NOTES:

The post synthesis simulation models incorporate some typical timing information from AMS. However, these are still not totally accurate. We should be in a position to carry out an even more accurate simulation at a later stage, in Lab 7.

Standard Project on VLSI Design SB1

Laboratory Guide 4 – Schematic capture & simulation with *Pyxis* Schematics & Eldo

This laboratory guide provides further experience in the use of the Mentor Graphics *Pyxis* Schematics package, emphasising its value for digital schematic capture and symbol creation. In this session you will create a schematic specification and a symbol corresponding to the ring oscillator element, **ring_oscillator**. You will then be able to use a similar approach to develop and verify the remaining parts of the design (already explored in concept using VHDL) relating to the programmable frequency synthesiser. Finally, you will create a component, named **top_level**, in which all the elements of the target design are embodied. When you complete the session, you should have a complete schematic representation for the programmable divider unit, which includes the ring oscillator design and all its other supporting elements, and you should have been able to verify correct operation of all parts using *Eldo*. Please note that this applies whether or not your other modules were generated by synthesis (using *Leonardo*) or by direct schematic entry.

Before you commence work with *Pyxis* Schematic, it is necessary to give some thought at this point to the form of the schematic sheet for the **ring_oscillator** module. This may usefully be done off-line, away from the workstation. Review the results of **Lab Session 2**, and, if necessary, consult the **Ring Oscillator Specification** pamphlet, which describes the system requirements for the ring oscillator. Then draw a rough sketch of your proposed schematic. Although the *appearance* of the schematic has no bearing on the functionality of the entity it represents, a neat and regular diagram clearly contributes to the process of understanding its operation. Show your plan to a demonstrator before you enter it into *Pyxis* Schematic.

Once this paper design is complete, you should log in at a workstation and start up *Pyxis* Project Manager, as described in the **Getting Started** pamphlet.

1. Start IC Studio

Use the steps detailed in **Lab Guide 1** and **Getting Started** to start up *Pyxis* Project Manager on your project. In the steps below we assume you have named the project library according to your, or your partner's **CRSid**; but if not, substitute in its place the **Shared-Library** name you chose.

2. Create a schematic for ring oscillator

With the **Shared-Library** you chose, give the command: **File > New > Cell** and enter the following details.

Cell Name : **RING_OSCILLATOR**

Click **OK** and a new cell will be created. Now, with the new cell highlighted in the **Object** window, right-click it and choose **New > Schematic**. In the **New Schematics** dialogue that follows, you can leave both text boxes as is, and just OK the box. A new schematic window will automatically be opened by *Pyxis* Schematic.

In this project, *Pyxis* Schematic has been configured with the AMS 0.35μm design kit, which enables access to some additional process-specific devices. The configuration is not complete, however, and we have to carry out a few work-arounds.

3. Place an instance of a 2 input NOR gate

Click on the **Add Instance** icon in the left toolbar. The **Enter Point** text bar will appear on the screen, and a File Browser pane appear.

Use this to navigate to the **CORELIB** Library in your project's list of resources. Left-click on the adjacent '+' sign to expand the list within, and scroll well down to **NOR20**. Left-click on this on to select that gate to place (or *instantiate*) on your schematic. Your cursor changes to a white symbol representing the device which you can deposit in a convenient place on the drawin area by left clicking. You will need to repeat this operation a number of times to obtain enough gates for your Ring Oscillator. A quick way to do this is to press **Ctrl+I**, which instantiates a further copy opf the same cell. Repeat the same step to place as many NOR gates as you need to create your design of ring oscillator. Alternatively, you may right-click on a NOR gate and choose **Copy** to duplicate the identical gate. You can also copy/paste entire rows of gates – but consider wiring some of them together before doing so.

4. Insert input and output ports

Use the **Add Port** icon at left to insert an input port; place it in a convenient position near your array of gates. Note that the same icon can provide **input**, **output**, and **bidirectional** (not required here) ports – click on the tiny pull-down arrow to see these options. Place an **output** port on your schematic. Repeat as many times as necessary to place all your ports on the schematic.

5. Wire the nets

Once all the gates and ports are placed, select **Add > Wire** to enter the wiring mode. This is indicated by the appearance of a mini **Add Wi**re dialogue at the bottom of the schematic window. The mouse cursor will also change to a cross.

Click on the starting point and double-click on the end point for each wire you would like to add. To make right-angle bends or 'jogs', single-click on each point where one is required. Once you have completed wiring the schematic, click **Cancel** on the mini dialogue or hit the **Esc** key to exit the wiring mode.

You will make mistakes when routing wires. There are a number of short-cut commands that can be used to edit and restructure wires; some are documented on CamTools, and the demonstrators will be happy to tell you about their favourites. Sometimes though the simplest option is to select the wire, and use the right-click menu to **Delete** it and restart.

6. Name the nets

The wires you have just added are known as **nets.** Every net must have a unique name. The name associated with each net is one of its **properties**. First of all, deselect all parts of the schematic by clicking on an empty location of the schematic – or hit **F2**. Next, select a specific wire to name by right-clicking on it.

From the pull-down menu **(NET) > Name Nets:** to call up the **Change Property Value** mini-dialogue for the purpose. Enter the following information.

Property Name	:	**NET**
Property Value	:	**<New net name>** (e.g. ENB, OUT1, OUT2)

Click **OK** to confirm it. The new name of the net connected to an input or output port will be reflected on the schematic. Repeat this step for the remaining wires.

Important: Take care to use consistent and meaningful names for nets, and be especially careful not to use the same name for different nets – *Pyxis* Schematics will not prevent this at the time, and although it is possible later to flag them up, it can lead to immense confusion later if they go undetected.

7. Allocate global power nets

All logic cells require power supplies and the schematic design process can help the designer ensure that proper provision has been made. This is important both in simulation and in preparation for layout.

You should be able to see that each of the cells you have placed has a reference to supplies: **vdd** and **gnd.** These are in effect reserved names. Note though that there is no obvious wiring to implement the necessary connections. The connections are implied, however, and these are referred to as **implied pins.** A part of this can be dealt with automatically through the use of **global nets**.

To ensure that all cells receive the necessary power supplies, and to make provision for the continuation of the local wiring to other cells, proceed as follows. Place two short separate lengths of interconnect in a convenient place at one edge of your schematic. They should not connect to each other or to anything else. Select them in turn, and use the **Name Nets** command to allocate them the names: **vdd** and **gnd**, respectively. This is to flag the need to make a connection to those points in a higher level of the hierarchy. That is all that is required at this stage. However, if this is not done, *Pyxis* will flag an error (see step 8) because no proper provision has been made for distribution of power. Moreover, simulations will likely fail (just as operation of a real circuit will fail if no power is provided).

8. Check the schematic

A schematic check is necessary to ensure that all connections are correctly wired. Choose **File > Check Schematic** to execute this check. A report window will appear to show any errors and warnings. Make changes in the schematic to fix all the errors.

Important: Be sure to discuss any unresolved errors or warning messages with the demonstrator. Unresolved messages may cause great difficulty and delay for you in completing the later stages of the project.

9. Save the schematic

Once the schematic is completed, you should save it by selecting **File > Save Sheet**. Of course, you should also carry out this step at regular intervals during any editing session to prevent the loss of your design if anything unexpected happens. Make a habit of executing a **Check Schematic** operation (see step 7)

immediately prior to saving. You can combine the two operations using **File > Check/Save**.

10. Generate a printout of schematic

To print out your schematic design, refer to the methods described in the **Getting Started** section.

11. Generate a symbol for schematic

Before this schematic can be used or called by another schematic, a symbol needs to be created for it. First of all, we ensure that the power requirements are propagated to the cell. Select the two pieces of wire interconnect you created as nets **vdd** and **gnd**. Then give the command: **Add > Generate Symbol** and a symbol generation dialogue will pop up.

Choose a suitable shape (or accept the default box format).

Click on the button: **Choose Implicit Pins** at the foot of the dialogue. The Set Implicit Pins should appear, ready populated with the two candidates you selected, **vdd** and **gnd**. Ensure the corresponding **Enable** boxes are checked, and OK the dialogue.

Finally, click **OK** to create a symbol.

Go to **File > Check Symbol,** and verify that there are no errors. There may be a small number of warnings, but these are usually acceptable. Give the command: **File > Save Symbol** to save the symbol you have generated. Investigate what another run of the Symbol Check reports.

Finally, go to **MGC > Exit** to close *Pyxis* Schematics.

Simulation of your design is performed using *Eldo*. Some preparation for this is required, but most of it goes on behind the scenes because the requirements are similar for all designs. *Eldo* cannot directly process the schematic files that comprise your ring oscillator design, and it requires a **device viewpoint** to be generated. A viewpoint acts as a set of rules to enable a tool to interpret schematics correctly. It also serves as a container object in which information relating to the simulation can be held. A design viewpoint, together with a component (i.e. a set of schematics designs) is formally termed an **electronic design configuration**.

12. Create a test schematic

Using the procedure described above (Step 2), create a test schematic with the following information, in the same library. This will create a new schematic design in *Pyxis* Schematics, and you should complete the dialogues as necessary to obtain the enter the following details.

Cell Name	:	**RING_OSCILLATOR_SIM**
Schematic Name	:	**Schematic**

13. Insert a single instance of the ring oscillator

Select **Add > Instance > Choose Symbol** and a dialogue will allow you to choose which symbol to add. Search for the **RING_OSCILLATOR** symbol you generated earlier in the **shared library**, and click **OK** to select it for placement.

14. Insert input and output ports

Use the same technique as in Step 4 to add input and output ports. Place them close to the input and output pins of the counter symbol in the test schematic.

15. Insert a DC power supply

You will need to repeat the same procedure carried out in step 7 to deal with power requirements. Place the necessary interconnect and apply the net names **vdd** and **gnd** in the same manner as before.

17. Wire all connections and rename ports

Referring to Steps 4 and 5 as necessary, make all necessary connections in your test schematic. Check and save your schematic using the procedures in Steps 8 and 9.

18. Prepare for simulation with *Eldo*

There is no need to leave *Pyxis* Schematics in order to carry out a straight-forward *Eldo* simulation, as the preparation and initiation of the simulation can all be controlled from within the Pyxis environment.

Click the green **Play** button at the bottom of the left toolbar and an **Entering Simulation Mode** dialogue will pop up. Click the New Configuration button; choose a name for this simulation to distinguish it from future simulations with different setups; for the moment, **Eldo-Run-1** might serve (NB: no spaces allowed). Click the radio button for **Eldo**, and OK the dialogue.

Pyxis Schematics will visibly change on entering this mode; the left toolbar will be replaced by simulation tools in place of drawing tools. In addition, the grids will disappear from the drawing area. A palette of handy macro-commands appears at the right hand side. No editing of the schematic is possible in this mode.

20. Check the setup of the simulation environment

None of the options under this heading is likely to need change, but you should explore the dialogue. Click on the green circular grid icon, and observe the compound dialogue box that appears. The **Environment Panel** allows you to select a number of different aspects to control. The **Netlister** gives fine control through a set of **Argument Switches** of how the details of the schematics are interpreted by *Eldo*, but the defaults are acceptable as they are for this introductory run. Note though that the default representation in *Eldo* for the ground connection is something called **GROUND** which is interpreted as **node** (or connection) **0**. This is a particular characteristic of *Eldo*, inherited from its **SPICE** forebears.

You should check that the **Simulator** is set to **Eldo** and the **Viewer** to **EZwave**, but other defaults may be also be left untouched. **Simulation Depository** determines where output files are sent, by default to the cell's directory.

21. Configure power supplies

The provision of power supplies, signals and so on in *Eldo* boils down to the specification of various electrical **sources.** For consistency with other simulators, however, it has become the convention to describe these as **forces**. Forces may be fixed or variable.

First, select the two pieces of interconnect designated as **vdd** and **gnd** by left clicking on each or by dragging a box across them with the left mouse button pressed. Then bring up the **Forces** dialogue. This can be done in at least two ways:

Either:

- Click on the **Forces/ICs** button in the palette menu at the right (choose **Manager)**; or:
- Click on the **ac/dc/tran** icon in the left toolbar, and click on **Forces** in the **Simulation Panel**.

Before we start assigning supplies, it is worth entering some symbolic parameters that can be used in place of individual numerical inputs. This allows for easy changing of supply voltages, etc, later.

Click now on **Params/Sweeps** in the **Simulation Panel**. Enter the following three parameters and their values in turn, in the corresponding boxes, leaving the **Type** set to **Global**.

PARAMETER	**VALUE**
p_gnd	0V
p_vdd	3.3V
inf	0

Pay attention to formatting in the above. The final parameter, **inf** is to work around some bugs in a number of cells in the library

Now return to the **Forces** aspect by clicking on that heading in the **Simulation Panel**. In the Selection from Schematic box, two nets are already present: they correspond to the ones selected on entry, **vdd** and **gnd** (note these are not case-sensitive. Click on each one in turn so it alone is highlighted, and choose **Source Type**: **DC**. For **GND**, set the magnitude to the string **p_gnd**; for **VDD** set it to **p_vdd**. Make sure the **Reference** is set to **GROUND** in each case; that **Voltage** is selected, and be sure to click the **Add** button each time so that both are transferred to the table at the foot.

Click **Apply**. Do not close the dialogue yet, as we have more settings to change.

As the dialogue closes you should observe circular markers appear on the schematic, denoting the power supply forces.

22. Set up transient analysis

With the **Setup Simulation** dialogue still active, choose **Analysis** in the **Simulation Panel**. In the Analysis Selector check the boxes for **OP** (Operating Point), **DC (**DC Analysis**)** and **TRAN** (Transient Response) – all of which abbreviations are part of the terminology for the industry-standard **SPICE** simulator. The other boxes are not normally checked for simulations of digital circuits.

Click on **TRAN** in the **Selector**. Set **Stop Time** to 1000n to give enough time for a few cycles of operation, and set **Print Time Interval** and **Max Time Step** to 1n. None of these is particularly critical, and you may wish to give thought to creating other configurations with a range of settings.

23. Set up a fixed force

Before running a simulation, signals or stimuli need to be provided to all input pins. In this test schematic, only **ENB** needs to be sourced with a signal. Highlight the wire connecting to the input pin and invoke the **Forces** dialogue again, as in step 21. On this occasion choose a **PULSE** as source. Be sure to change the **initial** and **pulsed** values to **p_vdd** or **p_gnd**, and adjust the various time parameters to configure a pulse that will hold the ring oscillator disabled for a while, then free-run.

Once you are content witb the signal values, click **Add**, then **Cancel**.

There are alternatives to the **PULSE** waveform approach for this purpose which you can research in the documentation.

24. Select signals to plot

Finally, you should deselect any existing nets by pressing **F2**.

Click on those nets that correspond to signals you would like to record for later plotting. These may be inputs, outputs, or a mixture.

Click on the **Outputs** button, and note that the nets you highlighted are listed in the **Selection from Schematic box. Note**: Multiple signals can be selected by holding down the Control key and clicking on the wires of interest.

From the drop-down **Task** menu, choose **PLOT.** Click **Add,** note the appearance of the signals in the list below (sometimes as a group), and click **Cancel** when content.

25. Run the simulation and review the results

You are now ready to run the simulation. Click on the green **Run Simulator** Play button. Observe any messages in the **Transcript Area**.

If all has gone well, your simulation should finish in a few seconds. Scroll back through the Transcript Messages to verify this, and when ready, clikc the **View Waves** button in the palette.

An *EZWave* window with generated signal waveforms should pop up within a few seconds. It should open the database of accumulated waveforms automatically, and you can the select which of the waveforms you wish to have

plotted. Zooming in and out of the waveforms can be done by using **View > Zoom In/Out,** or with the buttons provided in the *EZwave* toolbar above. To measure time differences, multiple cursors may be invoked by selecting **Cursor > Add**. The cursors can be flexibly dragged to different points or edges of interest. There is a tool for automatic waveform measurement that you may wish to investigate.

26. Export and print waveforms

To export the waveforms to a JPEG format file, choose **File > Export** and provide a relevant filename (e.g. **ring_oscillatorsim.jpg**). For printing, the instructions can be found in the section *Getting Started*.

Standard Project on VLSI Design SB1

IC Layout and Symbolic Representation

This pamphlet introduces the topic of IC layout in integrated circuit design and discusses the role of Design Rules and design rule checking (DRC) in the design process. It briefly describes the key features of Mentor Graphics' *Pyxis Layout* design package, which we shall use in this project to investigate the mask-level design of a CMOS 2-input NOR gate.

You may find it useful to refer to the pamphlet on **Design of Logic Gates in CMOS** which discusses some of the underlying theory.

Integrated Circuit Design Styles

The most detailed level of design specification for integrated circuits involves the specification of the materials from which they are manufactured, plus the fabrication procedures (typically oxidation, implantation, deposition and lithography). These are under the control of the **IC Process Engineers**.

At the other extreme, at the highest level of abstraction (sometimes called *front-end design*), the designer specifies system blocks, often in schematic form or using a VHDL-like language, and may have no knowledge of the underlying process.

Between the two extremes lie various strata of activities, some of which we have already investigated, including IC layout, layout verification and mask preparation, and simulation.

Full Custom Design

IC layout (or *full-custom* design) involves the specification of the lithographic masks used in the fabrication process to define the shapes and sizes of the conductors and semiconductors within the circuit. It is a style of design in which small details can have a dramatic effect on overall circuit performance, but which can also be the means of developing a compact and cost-effective layout.

Within the continuum of design activities over the IC industry as a whole, the proportion of layout designers (for digital designers at least) is actually quite small. This is largely because of the popularity of the *semi-custom* approach in which a library of *standard cells* is provided to fulfil the standard digital functions, combinational and sequential. This approach is also percolating through to linear/analogue system design.

IC Station

Pyxis Layout includes a collection of tools (including *Layout*, *Pyxis Assemble* and *Pyxis Router*) for use in physical layout of ICs. During physical layout, you build the mask data for the IC. You can handcraft each polygon forming the definition of the masks (sometimes known as '*polygon-pushing'*) in a full-custom approach, or you can use automated layout using standard cells and blocks of cells to assist you (semi-custom). Polygon layout generally gives you the most compact designs; a standard cell layout is generally faster to design, but is not as efficient in its usage of silicon. We shall investigate both approaches in this project.

In polygon layout, the designer makes use of a dedicated CAD editor package to draw and manipulate mask shapes. A typical CMOS process may require the definition of up to 12 different masks, all of which must be mutually compatible. The layout package must therefore display these simultaneously and clearly, and allow the designer to recognise and correct any adverse interactions. Mask shapes are displayed using a standard set of colour codes for clarity and consistency - for example the following colours are commonly used:

- Maroon Metal 1 (Aluminium interconnect)
- Yellow Polysilicon 1 (transistor gates & interconnect)
- Cyan + Brown Active region (n-type, for n-channel transistors)
- Cyan + Blue Active region (p-type, for p-channel transistors)

The editor provides a great many commands to add shapes and manipulate them in various ways. The majority of these involve selecting objects to be manipulated, which then become highlighted. A command is then invoked, and its action controlled by dialogues, prompts or signals from the mouse or keyboard.

*Pyxis Layout'*s layout editor has many features in common with *Pyxis Schematic*, and many of the short-cut keyboard commands work in a similar way. However, whereas in *Pyxis Schematic* we manipulate symbols or pre-designed schematics, in *Pyxis Layout*, the basic design objects are simply graphical shapes. Note that *Pyxis Layout* fully supports the concepts of *hierarchy*, in which a design may be specified at a high level in terms of subordinate modules or cells, which may in their turn be hierarchically specified.

Laboratory session 6 will include a number of exercises to help familiarise you with the practice of creating and modifying a mask layout using *Pyxis Layout* 's polygon editor.

Semi-custom design

Semi-custom design centres on the concept of a *library* of standard cells which are made available to a designer. In the simplest situations, a design will consist solely of a set of elements chosen from the library, organised where appropriate into blocks of circuitry. The only layout-related activities that are then required involve positioning of the cells or blocks in specific sites within the physical design (*Placement*), and the wiring together of these elements using interconnect in accordance with a schematic or other design (*Routing*).

Pyxis Layout provides a subset of commands for use in *Placement* and *Routing*. These may be carried out manually, or automatically. Manual placement and routing are generally tedious and time-consuming activities, but this approach may be essential for analogue or very high frequency designs where layout may have subtle

but important effects on performance. Automated layout may lead to a sub-optimal design (i.e. one in which space or the length of interconnections are not minimised), but the saving in time may be very significant. These commands actually summon up separate Mentor applications (*Pyxis Assemble* and *Pyxis Router* respectively), but this is effectively transparent to the user. With *Pyxis Layout* you can mix design methods within a cell, and change methods at any time.

Laboratory session 8 will introduce the use of the higher-level tools for semi-custom design, and will culminate in the development of a layout ready for submission to a fabrication foundry.

Design Rules

The IC designer usually wishes to make his designs as small as possible. Small transistors generally switch faster, consume less current, and - of course - occupy less space. As device sizes reduce, the implication is that a given system will fit onto a smaller and cheaper silicon chip, and will operate faster and with lower power consumption. All of these are admirable marketing attributes.

However, like any manufacturing process, IC fabrication is affected by *tolerances*. With a given set of manufacturing steps it is simply not possible to make satisfactory devices smaller than a certain size. As this critical dimension is passed, reliability falls sharply and the proportion of working devices at the end of the fabrication process also falls. The physical mechanisms for this are varied, but include vibration, thermal expansion, difficulty of maintaining precise conditions for chemical processing, optical diffraction in the photolithographic stages .. and several others. The process engineer will therefore wish to keep device dimensions sufficiently large that reliability and yield do not suffer.

Clearly a compromise is necessary between these conflicting demands. The result is a set of dimensional tolerances (or Design Rules) which the designer must observe in order that the design can be manufactured using a specific manufacturer's fabrication process. These can sometimes be simply expressed - for example, in the AMS process, the minimum dimension of polysilicon structures is stated as 0.35 μm. The minimum spacing between polysilicon structures is 0.45 μm.

The interpretation is simple: attempt to make polysilicon structures smaller than 0.35 μm, and you risk that they will exhibit breaks on fabrication; specify the minimum spacing less than 0.45 μm and you risk them shorting together. Other rules are specified in a more complex form, and vary with the context. There are over 100 rules applying to the AMS process in use for this project. An example - the set of rules for just one of its key layers, the polysilicon 1 layer **POLY1** - is attached as Appendix A to this document.

The variety of design rules in a typical process is such that the average human is quite unable to avoid inadvertently infringing them, especially when a primary objective in full-custom layout is to make designs as compact as possible. However, the workstation environment lends itself well to automating the procedure of detecting design rule violations. These can be highlighted on the screen with warning messages to assist the designer in correcting them.

It is normally required that all designs pass through design rule checking without violation before they can be fabricated.

Stick diagrams

One of the difficulties of full-custom layout is that the set of mask layers is, superficially at least, quite different in appearance from the original schematic. At the layout level, the boundaries between one device and the next may blur. Extra structures are often provided to satisfy process requirements, and the layout may be folded, re-dimensioned and warped in innumerable ways relative to the appearance of the schematic to achieve compactness, adjust parasitics, and generally to achieve the electrical performance sought.

For these reasons, a hybrid form of symbolic layout style is often used to represent cells. Like mask layout, this is often heavily colour-coded for visibility, but in the interests of simplicity omits selected layers (for example, semiconductor wells), and uses stylised structures for vias and contacts which ignore design rules.

Semiconductor devices are shown at unit dimensions (whereas in layout, correct dimensions would be arrived at by detailed design). Wires and transistors are arranged on a convenient grid. The resulting *stick* design is a valuable intermediate representation between schematic and layout, and allows the complexity of the physical structure to be concealed while assisting the layout designer to achieve a topologically correct result. We shall consider some simple schematics and the corresponding stick diagrams and IC layouts.

The inverter

By examining the schematic for the inverter, we can effect a physical layout by substituting layout symbols for the schematic symbols. In a schematic, lines drawn between device terminals represent connections. Connections between separate devices are dealt with simply by crossing lines - for example, the connection between the drain of the n-channel transistor and the drain of the p-channel transistor - see Figure 1(a) opposite. In a physical layout, however, we have to concern ourselves with the interaction of physically different interconnection layers.

We know that the source and drain of an n-channel transistor are fabricated from n-type material, while the p-channel device uses p-type semiconductor for these elements.

In CMOS technology, it is generally not possible to make a direct electrical connection from n-type material to p-type material (why not?). Thus we have to make the simple drain-to-drain connection in the physical domain with at least one conductive wire, in metal, and two contacts. Substituting layout symbols, the partial inverter in Figure 1 (b) results.

By similar reasoning, the simple connections to power, Vdd, and ground, Vss, are made using metal interconnect and contacts (c). The common gate connection may be a simple polysilicon strip or 'wire'. The resultant symbolic schematic is shown in Figure 1(d).

We continue the progressive conversion to a symbolic layout by representing the transistor devices as the superposition of polysilicon upon p-type or n-type active material. The schematic transistor symbols thus become unit rectangles of semiconductor material, yielding the symbolic layout or *stick diagram* shown in Figure 2(a), overleaf (to be completed).

Note that there is potential to vary the layout to suit particular requirements. In Figure 2(b), the transistors are aligned horizontally instead of vertically. Other arrangements

are possible if, for example, an inverter of **minimum height** but non-critical width were required to fit an existing design.

Similar reasoning can be applied to the 2-input NAND and NOR schematics. Figure 3 shows possible symbolic representations of these gates.

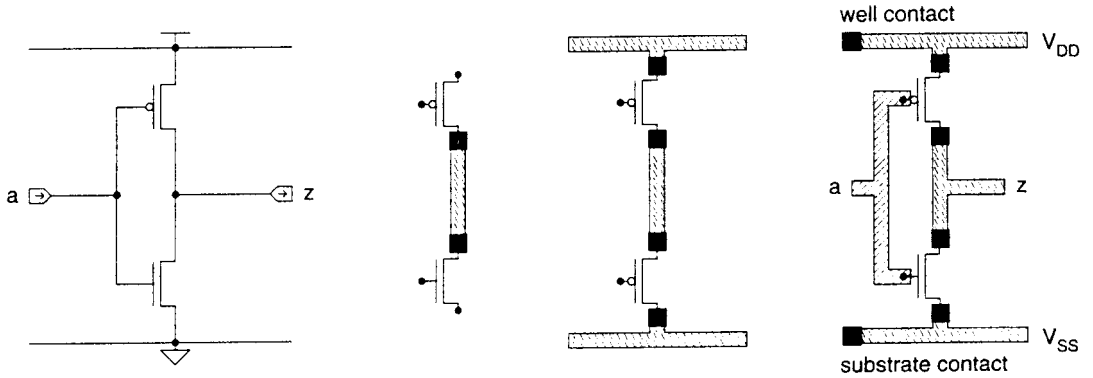
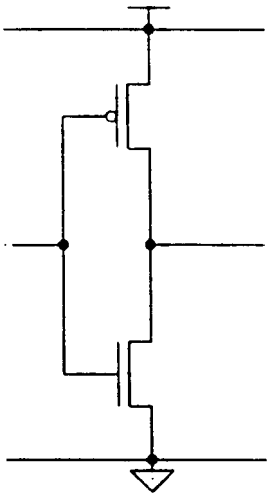


Figure 1 - From schematic to layout

Parametric Data

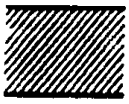
The materials and dimensions that are characteristic of a particular process also govern the electrical and other characteristics of the devices and other structures produced. For reference, some key parameters of the AMS 0.35 μm CMOS process are presented as Appendix B.

The transition from schematic to stick diagram (logic inverter)



Contact (black)

P-active (orange)

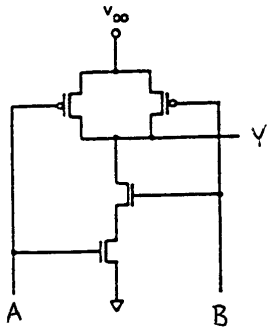


Polysilicon (red)

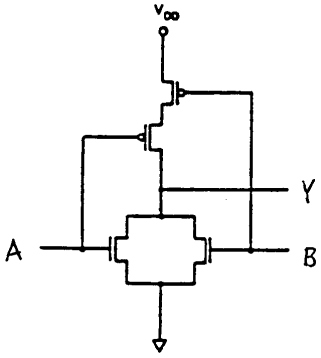
N-active (green)

Metal (blue)

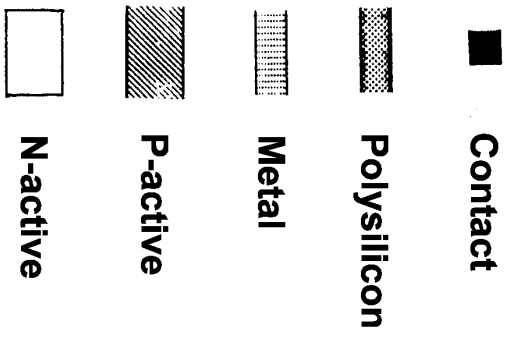
The transition from schematic to stick diagram (2-input gates)

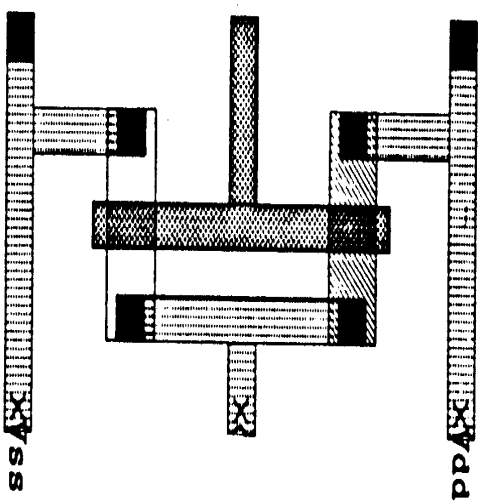


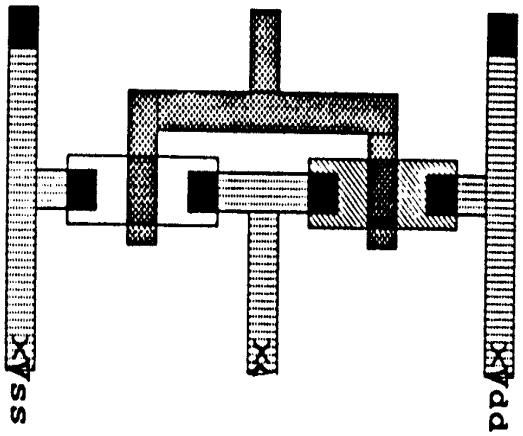
NAND2

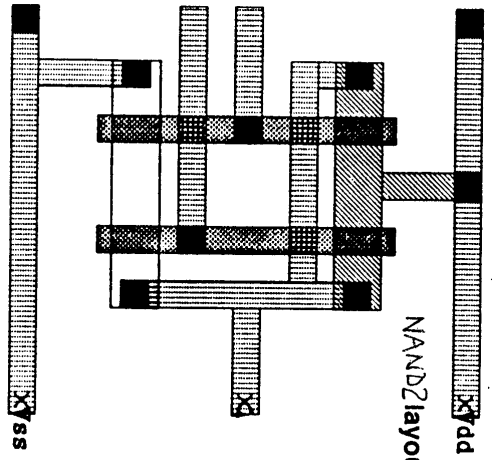


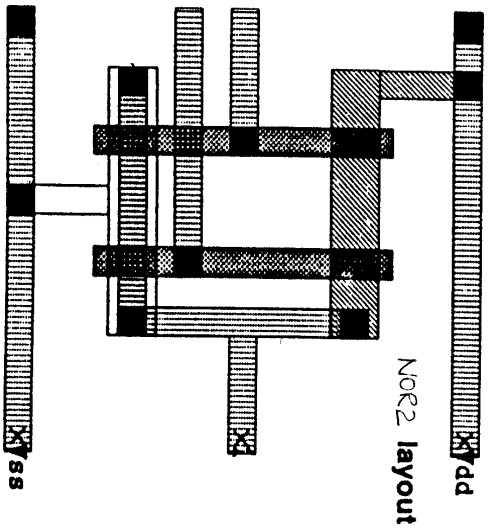
NOR2


N-active
P-active
Metal
Polysilicon
Contact


Vss
Vdd


Vss
Vdd


Vss
NAND2layout
Vdd

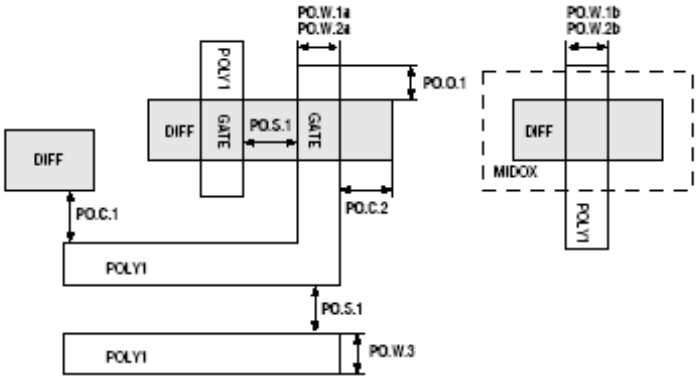

Vss
NOR2 layout
Vdd

Appendix A

4.1.3 POLY1

Rule	Description	Value [um]
PO.W.1a	Minimum GATE length of PMOS	0.35
PO.W.1b	Minimum GATE length of PMOSM	0.5
PO.W.2a	Minimum GATE length of NMOS	0.35
PO.W.2b	Minimum GATE length of NMOSM	0.5
PO.W.3	Minimum POLY1 width for interconnect	0.35
PO.S.1	Minimum POLY1 spacing	0.45
PO.C.1	Minimum POLY1 to DIFF spacing	0.2
PO.C.2	Minimum DIFF extension of GATE	0.5
PO.O.1	Minimum POLY1 extension of GATE	0.4
PO.R.1	Minimum density of POLY1 area [%] Density = total poly layer area / chip area Recommended dummy structures are 5um * 2um rectangles with 2um spacing. They should not be placed on active devices.	14
P1R002	POLY1 overlapping KEPOUT is not allowed	
S1KOP1	Minimum POLY1 spacing to KEPOUT or SFCDEF (not shown)	0.45

Guideline	Description	Value
G01P1	Maximum ratio of POLY1 area to connected CONT area	18000



Appendix B

Parametric Data for AMS 0.35 μm Process

Transistor					
	N-type		**P-type**		**Unit**
Gain factor (datasheets)	K_{Pn}	175	K_{Pp}	58	$\mu A/V^2$
Mobility (datasheets)	μ_n	0.038	μ_p	0.012	m^2/Vs
Gain factor (simulated)	K_{Pn}	115	K_{Pp}	40	$\mu A/V^2$
Threshold voltage (W/L=10/10)	V_{tn0}	0.46	V_{tp0}	–0.60	V
Threshold voltage (W/L=10/0.3)	V_{tn0}	0.48	V_{tp0}	–0.60	V
Effective channel length (0.3 μm)	$L_{eff;03;n}$	0.40	$L_{eff;03;p}$	0.53	μm
Effective channel length (0.6 μm)	$L_{eff;06;n}$	0.50	$L_{eff;06;p}$	0.50	μm
Body effect factor (W/L=10/10)	γ_n	0.58	γ_p	–0.45	√V
Resistance, active region (sim.)	r_{dsn}	55	r_{dsp}	55	ΩA/μm
Saturation current (0:3 μm)	I_{satn}	540	I_{satp}	–240	μA/μm
D-S breakdown volt. (0:3 μm)	V_{brn}	> 8	V_{brp}	> –8	V

Sheet resistance		
Layer		Ω/□
metal4	*Rsm*4	0.05
metal3	*Rsm*3	0.05
metal2	*Rsm*2	0.08
metal1	*Rsm*1	0.08
poly1	*Rsp*	6
poly2	*Rsp*2	50
n^+ diff.	*Rsdn*	80
p^+ diff.	*Rsdp*	150

Capacitances (layer to substrate)				
	Area		**Perimeter**	
	fF/μm^2		fF/μm	
gate capacitance	C_{ox}	4.60		
gate-diff overlap			C_{gd0}	0.21
gate-bulk overlap			C_{gb0}	0.11
n+ diffusion (0 V)	C_{jn}	0.93	C_{jnp}	0.28
p+ diffusion (0 V)	C_{jp}	1.42	C_{jpp}	0.38
Nwell − bulk (0 V)	C_{jw}	0.11	C_{jwp}	0.53
poly1	C_{p1}	0.119	C_{pp}	0.052
metal1	C_{m1}	0.032	C_{m1p}	0.046
metal2	C_{m2}	0.012	C_{m2p}	0.036
metal3	C_{m3}	0.008	C_{m3p}	0.037
metal4	C_{m4}	0.006	C_{m4p}	0.033
poly1-poly2	C_{polys}	0.86	C_{polyp}	0.082

Max. current density		
Layer		mA/μm
metal4	*Jm*4	1.6
metal3	*Jm*3	1.0
metal2	*Jm*2	1.0
metal1	*Jm*1	1.0
poly1	*Jp*	0.5
poly2	*Jp*2	0.3

Diode data		N	P
Area junc. pot.	V_j	0.69 V	1.02 V
Sidewall junc. pot.	V_{sw}	0.69 V	1.02 V
Area grading coeff.	m_j	0.31	0.55
Sidewall grading coeff.	m_{sw}	0.19	0.39

Parametric Data for AMS 0.35 μm Process (cont)

Max. contact current		
0.4 μm×0.4 μm contact 0.5 μm×0.5 μm via, via2, via3		
Layer-layer		mA
metal4-metal3	I_{via3}	0.96
metal3-metal2	I_{via2}	0.60
metal2-metal1	I_{via}	0.60
metal1-poly1/diff	I_{cp}	0.94

Contact resistance		
Layer-layer		Ω/cnt
metal4-metal3	*Rvia*3	3
metal3-metal2	*Rvia*2	1.5
metal2-metal1	*Rvia*	1.5
metal1-poly1	*Rcp*	5
metal1-n+ diff. .	*Rcdn*	40
metal1-p+ diff.	*Rcdp*	90

Structural and geometrical parameters		
Gate oxide thickness	t_{ox}	7.5 nm
Poly1-poly2 oxide thickness	t_{pox}	41 nm
Field oxide thickness	t_{fox}	290 nm
Poly1-metal1 oxide thickness	t_{pox}	645 nm
Metal1-metal2 oxide thickness	t_{mox}	1.00 μm
Metal2-metal3 oxide thickness	t_{mox2}	1.00 μm
Metal3-metal4 oxide thickness	t_{mox3}	1.00 μm
Passivation thickness	t_{prot}	900 nm
Poly1 thickness	t_p	282 nm
Metal1 thickness	t_{m1}	665 nm
Metal2 thickness	t_{m2}	640 nm
Metal3 thickness	t_{m3}	925 nm
Metal4 thickness	t_{m4}	925 nm
n^+ and p^+ junction depth	x_j	200 nm
n-well junction depth	x_w	2.0 μm

Standard Project in VLSI Design SB1

Laboratory Guide 5 - IC layout with *Pyxis Layout*

This laboratory guide provides an introduction to full-custom IC layout using Mentor Graphics' *Pyxis Layout* tool. The procedures in this session will lead to the creation of a layout for the 2-input NOR gate **nor2x** which we shall investigate in comparison with the AMS library version of that gate. The ultimate objective will be to obtain a neat, compact layout, free from design rule violations, with the fastest possible rise and fall times at the output terminal.

The starting point is a part-complete version of the required **nor2x** layout. This is provided within the external library **sb1_nor2_lib**, which you will find in your cbt directory. A check-plot and circuit schematic are attached as an appendix to this guide. The layout supplied determines certain key aspects of the design - for example, the pitch between the metal interconnect lines corresponding to the supply rails, and the positions of the two inputs and the output - to ensure compatibility with other gates and cells used in the design. However, as provided, the **nor2x** layout has no polysilicon transistor gate electrodes, contact structures are displaced, and some interconnect is omitted. In the later sections, you will remedy these omissions, adapt and enhance the design, and check that the resultant layout does not violate the applicable design rules. Before you begin, however, you should carry out the following preparatory exercises. You should include the resulting material in your final report.

Most of the following activities do not require the use of the workstation. You need to complete these before you commence work on IC layout.

Consider the **nor2x** transistor level schematic representation given at the end of this session guide. Use it to construct a simple stick diagram for the **nor2x** gate, in the manner shown in the lecture. Compare your diagram with the given **nor2x** layout, and determine the information listed below.

- Identify the positions of the various input and output ports on the layout.
- Determine the position of the supply rails, and use the ruler provided on the check-plot to estimate their width and separation. The dimensions chosen for these structures are closely dependent on the current consumption of the gate.
- The supplied **nor2x** layout does not indicate the position of the gate, source and drain electrodes for the various transistors. Study the **nor2x** schematic and layout, and, taking into account the required transistor dimensions, attempt to determine the most suitable positions for these electrodes. You may find it helpful to refer to the notes on IC Layout and Symbolic Representation. Note also that there is no unique answer to this question, as a number of different possible layout styles are likely to be suitable.
- Annotate the **nor2x** layout provided with the intended source and drain positions, and the sites where the polysilicon gate electrodes are required.
- Consider how to modify the layout to implement an electrical connection between the drain electrodes you have proposed, and the output port Y of the gate.

Discuss your conclusions with a demonstrator before you start the activity.

In the following activity, you will modify the part-completed nor2x layout to make it suitable for incorporation into your ring design.

You should be logged in at a workstation as described in the **Getting Started** pamphlet, with *Pyxis* Project Manager running. The current working directory should be **$CBT_WD** (which is the soft prefix for your **$MHOME/cbt** directory), and you should see your project in the Navigator pane. The external library **sb1_nor2_lib** should also be visible in the Navigator, but outside your project.

1. Create an internal library to receive the contents sb1_nor2_lib

Open the **sb1_nor2_lib** library by left-clicking on the associated '+' sign in the Navigator pane. You should be able to see the three elements comprising the nor2x cell: **layout**, **schematic** and **symbol**. The icons used are largely self-explanatory, but the *Pyxis Schematic* Quick Reference (see CamTools) contains a handy summary of these.

It would be possible to work on these items *in situ*, but it is better practice to use these as a reference and import copies into an internal library.

With *Pyxis* Project Manager activated, select your current project by left-clicking once. Then, press the right mouse button and in the drop-down menu that results, select **Library.** In the dialogue that results, enter a suitable name, for example, **LayoutLib**. This will create a new internal library within your project.

Now open up the external library **sb1_nor2_lib** by left-clicking its entry in the **Navigator** pane. With the contents visible, click once on the **nor2x** container, and use the right mouse button **(RMB)** to choose **Copy**, in order to copy the contents. Click on the new internal library, just created, to open it, and use **(RMB) > Paste** to paste the **nor2x** content locally.

2. Review the nor2x library

In *Pyxis* Project Manager, open up the new library, if necessary, by left-clicking on it. The upper right-hand pane should show the three design items it holds, of type: **layout**, **schematic** and **symbol** respectively. We shall need to interact with the **schematic** and (especially) the **layout** views. Note that while these parts are not qualified items from the AMS library, they are based on the same AMS process and it should be possible to incorporate them (on completion) into your design alongside regular AMS library components. The layout is a part-complete representation of the mask-level physical layout specification of a 2 input NOR gate, which you will complete.

First, select the schematic view by right clicking it; use the resulting menu to select **Open** > **Schematic Editor** This will open up *Pyxis* Schematics on the schematic.

You should see a transistor-level representation, from which you can identify the four component transistors, types **pmos4** and **nmos4**, identified as instances: MP1, MP2, MN1 and MN2.

3. Study the characteristics of the devices used

Select each MOS transistor in turn, and use the **Report** > **Selected Object** command to display the properties associated with the cell. The property names in **l** and **w** represent the length and width of the corresponding transistor channels

in micrometres. Record these. You can alternatively select all devices and see the properties in one report.

Show that the values of L and W are consistent with the device mobilities μn and μp. (See the **IC Layout and Symbolic Representation** section)

4. Preparing for layout

Return to your *Pyxis* Project Manager window, and with the **nor2x** member of the library highlighted, right-click the **layout** icon and select the **Open > Layout Editor** option. *Pyxis* Layout should start up, with the layout cell visible. Drag and adjust the *Pyxis* Layout window to fill the screen conveniently. Use the short-cut key **Shift + F8** if necessary to have the layout fill the drawing window.

During this session you will repeatedly be selecting shapes, items and assemblies of these with the mouse. There are many options available for streamlining the user interface for activities of this kind, as with other Mentor packages. You may wish to spend a moment reviewing these – to access them, give the command: **Setup > Preferences**, and look in particular at the **Selection** options by clicking on the corresponding icon at the top.

5. Add rulers to the layout to assist in sizing and placing objects

The partial cell design as supplied has a coordinate origin at the lower left corner: note the cross symbol which signifies this. Give the command:

(Menu bar) > Add > Ruler.

When the prompt bar appears, leave the settings at their defaults. Observing the cursor coordinates provided centre-screen on the status bar, place the cursor at a horizontal coordinate of -4, and a vertical coordinate of around -1. At these coordinates, press and hold down the Select mouse button, and drag the resultant line horizontally to the right until it reaches a horizontal coordinate of 12 or so. Repeat this operation, but this time create a ruler starting at (-4,-1) and extending vertically for the height of the design.

If you make a mistake creating rulers, you can delete them all using the command:

(Menu bar) > Edit > Delete > Rulers All.

Alternatively, select the offending ruler and use the command: **Edit > Delete Selected** to remove it.

6. Keeping the cell reserved for Edit operations

Since Mentor *Pyxis* is a multi-user system, the possibility of two designers simultaneously attempting to edit a design must be avoided. Once a cell has been reserved by one designer, it cannot edited by any other user (or even the same designer in a separate *Pyxis* Layout session) until the original edit is closed. On entry to *Pyxis* Layout, assuming your partner (or you yourself) have not opened a previous editing session on the cell, it will be automatically reserved for edits. If editing commands are largely greyed out, the probability is that an earlier session is still running. If you find yourself in this situation, consult a demonstrator.

In this project, each **mentor***x* user has a private copy of the original **nor2** cell rather than just one master copy, to avoid this kind of contention.

7. Change the Visible Layers seen on the screen

Examining the **nor2x** layout shows that it consists of a number of colour-coded superimposed shapes or layers. As explained, the masks used in the various stages of the fabrication procedure are each represented by a colour-coded set of shapes. For the purposes of carrying out the design, it is vital to be able to visualise the complete mask set, to ensure that each mask is consistent with the remainder. However, this can lead to a rather complex cluttered appearance. Sometimes it is helpful to suppress display of certain kinds of shapes.

To experiment with this capability of *Pyxis* Layout, you will need to intereact with the **Layer Palette** normally seen by default near the right of the screen. If it is not visible, give the following command: **(Menu bar) > Setup > Windows**, and note the item: **Layer Palette** on the list. This needs to be checked by clicking on the item; if it is already, no action is needed.

The **Layer Palette** should now be visible. It shows a list of the available layers, each individually colour coded, and a corresponding array of checkable boxes organised in three columns, labelled S (selectability), V (visibility) and F (fill). A scroll bars allows you to scroll up and down the list, which you should do, to get an impression of the kinds of layers available in this design, and how they look Some appear to be duplicates, but they have different functions. Right-click in the header bar and check the **Purpose** item to see what they are used for.

Now click the action box containing a '-' sign above the **V** column; this should make all layers invisible. In turn, check the V column boxes for the three layers: **POLY1**, **CONT**, and **MET1**. By default the screen should update at once; if it does not, click the **U** (update) icon above the vertical scroll bar in the Layer Palette.

You should see that the resultant display is considerably simpler: only conducting (as opposed to semi-conducting) layers are now visible. Note that this has not changed the design in any way; it has simply made parts of it invisible.

Restore display of all layers by clicking the action box (with the '-' sign) so it contains a tick. All layers should be visible once again.

You may find it convenient to restrict visibility of the layers at times, in the way just explored, so you can see more clearly when carrying out the modifications described below.

8. Investigate the basic commands for placing shapes

Experiment with the keystroke commands +/– (on the Numeric Keypad) and **Shift+F8**, which control the display zoom factor, and note the scroll bars at bottom and right. Organise the display so that there is a substantial clear region to left and right of the **nor2** structure.

Click on the **POLY1** item in the Layer Palette so it becomes highlighted. Give the command: **(Menu Bar) > Add > Polygon**. This allows you to add a rectangle or polygonal shape to the layout. With the cursor over a clear area of the design window, and with the left mouse button depressed, drag out a rectangle. Note that the size of the rectangle is read out in the status bar. Release the button when the rectangle is about 4 μm by 2 μm, and observe that it is displayed *selected*, in white. Observe also the bold white icon similar to an hourglass at one corner of

the shape. This is the *basepoint*. When you **Copy** or **Move** selected objects, they are placed so that when you identify the destination point with the mouse, the basepoint is placed here, and all other elements of the selection are placed relative to it. Click **F2** to **Unselect All**, and note that the shape is displayed with its designated colour and fill-style.

Repeat the **(Menu Bar) > Objects > Add > Shape** command, leaving the options at their defaults. However, on this occasion, instead of dragging out a rectangle, define a closed polygon in the shape of a **L** - in a clear area of the design, by clicking the Select button at its vertices and double-clicking at the end point (which should coincide with the start point). The shape outlined should be displayed, selected, as polysilicon1 (**POLY1**).

Using the non-rectangular shape you've just created, experiment with **(Menu bar) > Edit > Move > Selected** and **(Menu bar) > Edit > Copy > Selected, (Menu bar) > Edit > Flip** and **(Menu bar) > Edit > Rotate**, using them to move your shape to a different position, create a duplicate, and manipulate it in various ways.

When you finish with these commands, press **F2** to de-select all objects.

9. Investigate the Notch and Stretch commands for editing shapes

Now investigate the **Notch** command. De-select all shapes by pressing **F2**. Select just one of your shapes by clicking it. Give the **(Menu bar) > Edit > Notch** command. When the prompt bar appears, place the cursor within your shape, and using the Select button, drag out a rectangular region which extends outside the shape. Release the Select button. Observe the results.

Repeat the command, but on this occasion begin dragging outside your shape and allow the rectangle being dragged out to enter the shape. Experiment further with this command if you wish.

Now investigate the **Stretch** command. Ensure just one of your shapes is selected, and give the command: **(Menu bar) > Edit > Stretch**. Drag out a rectangle (using the Select button) which fully encloses *one* edge of the selected shape. The white rectangle should persist; now drag out a line starting at a point within the white rectangle, and roughly perpendicular to the chosen edge, and finishing outside the box. Release the Select button, and observe the results. Experiment further with **Stretch** if you wish.

Now delete the two experimental shapes you have created by selecting them (only), and using the **(IC Window) > Edit > Delete** command.

In order to ensure you start with a clean copy of **nor2x**, close the cell now by clicking the X system button at the top left of the design window. When the **Close Window – Check Cells to Save** dialog box appears, uncheck the box adjacent to **nor2x**, because you do not wish to preserve the experimental shapes as part of your layout. OK the box. Re-open the **nor2x** cell by giving the command:

(Menu bar) > File > Recent

and with the left button held down, select the **nor2x** cell again.

10. Run *ICrules* to check for any existing design rule violations

The importance of ensuring that any mask layout comply with the manufacturer's design rules has been mentioned. When a layout is being edited manually, it is

necessary periodically to run a check that none of the shapes laid down violates the set of rules. This is done by means of *ICrules*, an interactive tool controlled from within *IC Station*, which detects and highlights any rule violations. *ICrules* allows a quick and interactive means of detecting basic errors in layout, though it is not really adequate ensuring the correctness of complete design. A higher performance, independent program *Calibre* is available for this. In fact, *ICrules* invokes *Calibre* in a special mode to carry out the quick quick check. We shall begin by carrying out comparatively straightforward interactive checks on our design, but, in accordance with industry practice, we may follow this with a full-scale review using *Calibre* once the design is complete (if time permits).

As provided, the **nor2x** cell is only partially laid out and has a few design rule violations which you are asked to identify and correct as part of your design activity. Some of these arise because the two isolated structures created to the left (which are contacts **CONT** to **METAL1**) have not been properly integrated into the design. To run the design rule check (DRC), click the **Instant DRC** entry in the **IC palette**, which switches to display the Instant DRC, the one required to run *ICrules* in interactive mode. The **nor2x** design is automatically set up to use the correct rule set (AMS **HK_C35**).

Please note that the alternative **IC Palette** option **ICrules** is not currently set up. It is also possible to uset the command **(Menu bar) > Tools > Calibre** to access *Calibre* in its stand-alone mode, but this is not recommended as it operates outside the *Pyxis* Layout environment.

To begin the check, click the **Check** palette menu item. It is also possible to use the **Check** Area button. This will allow you to specify a rectangular area (using the mouse select button to drag out an area) in which the check is to be confined. In this case we shall leave the area unspecified, causing *ICrules* to check the entire design. With **Check** clicked, observe the Message Area.

You may observe some warning messages flash past. When the check is complete, any warnings or errors are gathered into a database, and you should see a text box announce how many rule violations have been found. The first is highlighted when you OK the text box, or if you later click the **First** item in the palette.

You should observe a portion of the design flash three times, then remain illuminated. This identifies the site of the error. The description is presented in the Message Area, and you may get further enlightenment by studying the human-readable version available as a PDF – see CamTools, Project Documents, *AMS C35 0.35 um CMOS Design Rules*. Study the layout and the error description carefully, and try to identify the correction(s) required. Some may be a little misleading, because the layout is incomplete. Consult a demonstrator if you are unsure. You can make changes to the layout as you go through the list, to take advantage of the graphical prompts provided by *ICrules*. When you are ready, move to the next violation by clicking the **Next** item. Note that some of the errors mentioned are quite benign, and may not need correction – ask a demonstrator if in doubt. When you have later made all the corrections you can, re-check the design once again and save the cell by giving the command:

(Menu bar) > File > Save > Layout.

11. Lay out the gate electrode for input A

Referring to your stick diagram and annotated layout, decide where to place the polysilicon gates to form the n- and p-type transistors required in the **nor2** design to connect to input **A**. Both **A** and **B** ports need to be placed somewhere in the middle of the design, and must expose a connection in metal1, to allow a connection to be made to other parts of the circuit. A previous designer has kindly left suitable structures lying in the cell – the two square structures to the left. Closer examination will show these have ports **A** and **B** already associated with them.

You will need to integrate these into your design by moving them to appropriate positions. Broadly speaking, you will need to create a set of vertical strips in **POLY1** - though not necessarily of uniform width or in straight lines - connected to the corresponding input port and crossing two active regions. Be careful that the layout you choose for **A** will not impede you in designing a similar structure for input **B**.

Use the **(Menu bar) > Add > Polygon** command investigated above to generate the gate connected to input port **A**. You may have to do this in a number of stages, using abutted or overlapped rectangles of different sizes. Use *Instant DRC* at regular intervals to verify that the structures you design are free from violations; and note that you can restrict its operation to a restricted rectangular area. Although this may seem irksome, it is probably more efficient in the long run than laying down an entire complex structure and having to wade through a plethora of violation messages at the end. The demonstrators will help you interpret any less obvious warning/error messages.

12. Lay out the gate electrode for input B

Referring to your stick diagram and annotated layout, decide now where to place the polysilicon gates corresponding to input **B** (also in the central part of the design). Again, a vertically oriented strip of polysilicon is required. On this occasion, instead of laying out a set of abutting shapes, we recommend you use *Pyxis's* facility for laying out **path** structures. With this command, you determine the position of the centre line of a rectangular shape, which may have jogs and changes of direction if required; software then 'fleshes out' the centre line to the required width. **Path** structures can as a result be recognised by the existence of a prominent centre line.

The default path widthe drawn by *Pyxis* Layout is 1μm, which is too wide for this application. Determine a suitable width for the path, noting that this will be constant over the whole of its length. There is no convenient dialogue box that allows you to preset the path-width, but it can readily be done by a typed command. Suppose you wish to make the path width 1.2 μm. With the cursor somewhere over the layout area, type the following at the keyboard:

```
$set_ic_path_width(1.2)
```

followed by **Enter**. Note that as soon as you type '$', a small text box opens up on the screen to receive the rest of the command.

Now, in the **Layer Palette**, click once on **POLY1** to highlight it. Give the command: **(Menu bar) > Add > Path**. When the cursor transforms into a cross, using the left mouse button, mark the position of the start point and, in order, any

other vertices required, double clicking at the end point. The path will then be displayed, filled out to the prescribed width, and selected.

If you are unhappy with the course of the path, perhaps because it violates a design rule, or for any other reason, you may delete it like any other shape. Alternatively, you can use **Stretch** to extend a segment.

With the path selected, you can use the **Object Editor** dialogue to change the width of the path along its entire length. To do this, give the command: **(Menu bar) > Setup > Windows** and check the **Object Editor** item in the pull-down menu that results. Then, in the **Object Editor** dialogue, edit the **Width** parameter as required.

You can also select a segment of the **path** itself, by giving the command: **(Menu bar) > Select > Select > Edge:**, then clicking the Select button with the cursor over the centre of the segment in question. The **(Menu bar) > Edit > Move** command may then be used to reposition the path as required.

As an alternative, with the whole of the path selected, you can use **(Menu bar) > Edit > Modify Ctr. Line** to continue the path. When you are satisfied, check that *InstantDRC* runs clean, and save the design.

You should take care that your polysilicon gate electrodes have the correct dimensions so they correctly define the length of the conductive channel. They must also satisfy the design rule requirements, but you will only achieve the anticipated electrical performance if the transistor dimensions are chosen correctly and laid out in accordance with your earlier calculations. Edit if necessary.

13. Perform advanced Design-Rule Check with Calibre (optional)

Once the design is completed to your satisfaction and that of *InstantDRC*, you may wish to invoke **(Menu bar) > Tools > Calibre > Run DRC** to start up Calibre for design rule checking. The proper **Calibre-DRC Rules File** should already be set:

$AMS_DIR/tech/HK_C35/rules/calibre/c35b4/c35b4c3.rules

It is possible to select the checks that you wish to perform by clicking on the **Rules** button and entering the **Selection Recipe Edit** feature, using **Setup > Select Checks** from within Calibre. The available checks include the following collections:

```
GROUPS
Checks by group
LAYERS
Checks with selected layer only
Checks including selected layer
CONNECTIVITY
Checks with connectivity
Checks without connectivity
DENSITY
Checks with density
Checks without density
```

You may de-select any checks that are not necessary. It is essential that all rules are passed for any final top level design and higher-level sub-cells, if it is

intended for fabrication. But for a lower-level cell, many of the checks can be ignored.

After passing these standard checks, additional checks can also be performed. These require different rule files which can be loaded instead:

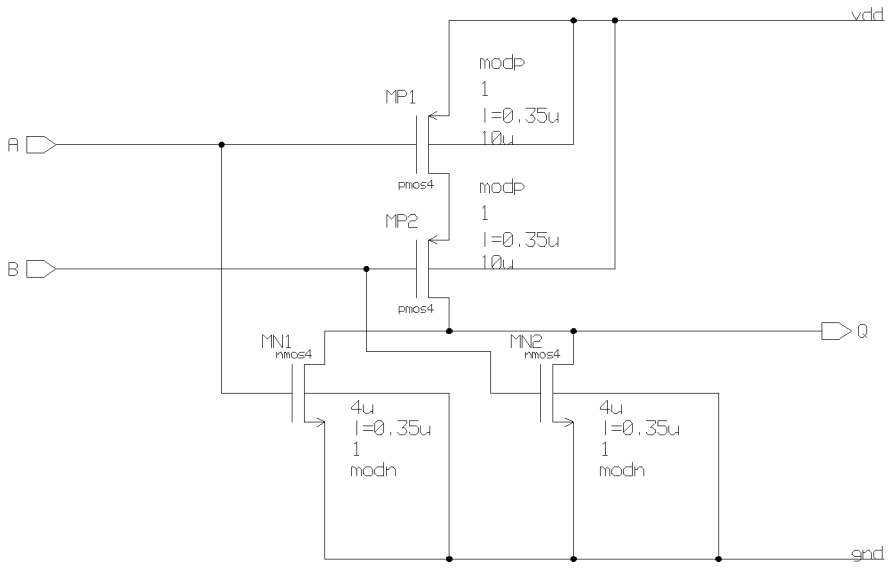
```
$AMS_DIR/tech/HK_35/rules/calibre/c35b4/c35b4_antenna.rules
$AMS_DIR/tech/HK_35/rules/calibre/c35b4/c35b4_esd.rules
```

Click **Run DRC** to perform checking. At this point, the *RVE* window is llikely to pop up with a number of errors. Double-click on any specific error and the coordinate location to get more information about it. The part of the layout that needs to be fixed will be highlighted in *Pyxis Layout***.**

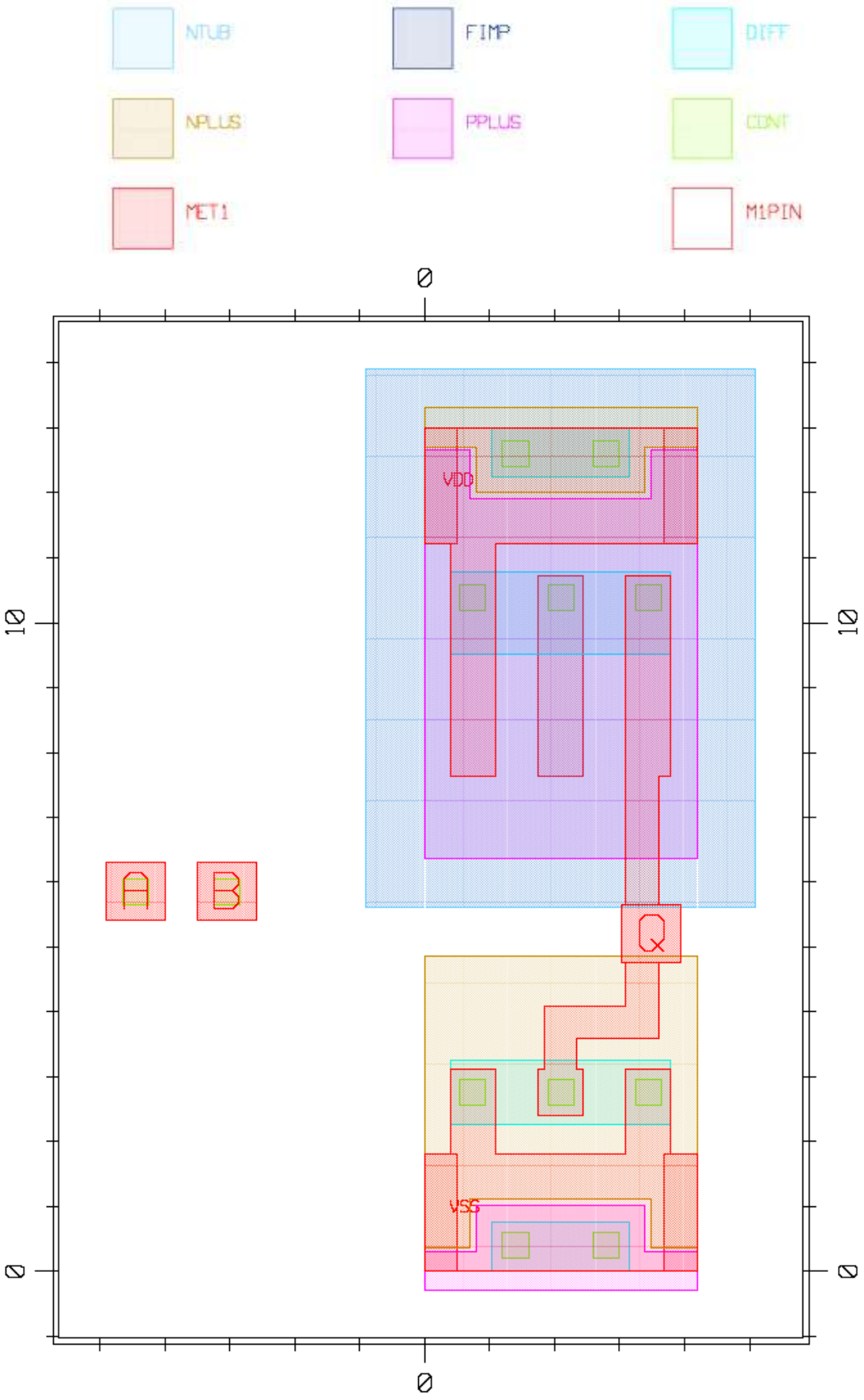
14. Generate a checkplot

You will probably wish to generate a checkplot of your completed **nor2** layout. At this stage a monochrome laser printout should serve your needs, but you may wish to annotate or colour-code the result for inclusion in your report in the interests of clarity. The procedure for generating hardcopy of layout files is described in the **Getting Started** section of your Project Guide.

Note that in order to print the cell, you must have it visible on the screen, and the design window must be active.



Schematic representation of **NOR2X** logic gate



Layout of **nor2x** cell prior to modification
Dimensions of the rectangular bounding box are approximately 14 µm × 7 µm
[A colour version of this plot is available on CamTools]

Standard Project in VLSI Design SB1

Mixed-signal Simulation and Eldo

This pamphlet briefly describes the role of device-level simulation in integrated circuit design and introduces the Mentor Graphics *Eldo* simulator. *Eldo* is largely based on the industry-standard SPICE simulation package developed at Berkeley University. *Eldo* is a member of the Mentor Graphics *Questa ADMS* simulation suite, which incorporates several compatible digital and HDL-based modelling tools; circuits can be a complex mix of analogue and digital devices, with models of many different types.

Eldo is a mixed-signal circuit simulator that allows you to verify the functionality of analogue/digital electronic designs produced with *Pyxis Schematic*. *Eldo* offers:

- interactive control of the simulation which includes the facility to let you apply signals, list them, run the simulation with chosen constraints, and create charts of results
- the ability to save setup conditions, stimuli, results and particular states of the simulation so they can be restored at any future time
- incremental design capabilities that let you update property values during the simulation, allowing you to use heuristic or 'what if' methods for improving the design you create
- compatibility at the interface level with Mentor full-custom and verification tools allowing layout parasitics to be incorporated in the simulation
- a substantial library of standard component models, pre-characterised and identified by industry-standard part names. For example, if your design calls for a 741 op-amp, you can specify that the model for this device be used in your simulation
- compatibility with the industry-standard SPICE simulator, from which *Eldo* was developed.

Eldo's great strength is that it can perform a full numerical analysis of the voltages and currents appearing at every node of the circuit being simulated. *Eldo* is capable of DC analyses, DC sweeps (allowing transfer functions to be determined), AC analyses (including frequency response and noise analysis), transient response, and a number of others. For logic gate simulation, transient responses and DC analyses and sweeps tend to be of greater use. Stimuli can be expressed as DC voltages or currents, sinusoids, exponentials, pulses or piece-wise linear waveforms. The numerical models used may be elaborate - for example, in the case of MOS transistors it may be necessary to specify 40 or more parameters per device.

For example, a representative model used to describe MOSFETs similar to those in the AMS 0.35 μm process is shown below:

```
.MODEL CMOSN NMOS (                                    LEVEL   = 3
+TOX     = 7.9E-9          NSUB    = 1E17              GAMMA   = 0.5827871
+PHI     = 0.7             VTO     = 0.5445549         DELTA   = 0
+UO      = 436.256147      ETA     = 0                 THETA   = 0.1749684
+KP      = 2.055786E-4     VMAX    = 8.309444E4        KAPPA   = 0.2574081
+RSH     = 0.0559398       NFS     = 1E12              TPG     = 1
+XJ      = 3E-7            LD      = 3.162278E-11      WD      = 7.046724E-8
+CGDO    = 2.82E-10        CGSO    = 2.82E-10          CGBO    = 1E-10
+CJ      = 1E-3            PB      = 0.9758533         MJ      = 0.3448504
+CJSW    = 3.777852E-10    MJSW    = 0.3508721          )
.MODEL CMOSP PMOS (                                     LEVEL  = 3
+ TOX    = 7.9E-9           NSUB   = 1E17              GAMMA  = 0.4083894
+ PHI    = 0.7              VTO    = -0.7140674        DELTA  = 0
+ UO     = 212.2319801      ETA    = 9.999762E-4       THETA  = 0.2020774
+ KP     = 6.733755E-5      VMAX   = 1.181551E5        KAPPA  = 1.5
+ RSH    = 30.0712458       NFS    = 1E12              TPG    = -1
+ XJ     = 2E-7             LD     = 5.000001E-13      WD     = 1.249872E-7
+ CGDO   = 3.09E-10         CGSO   = 3.09E-10          CGBO   = 1E-10
+ CJ     = 1.419508E-3      PB     = 0.8152753         MJ     = 0.5
+ CJSW   = 4.813504E-10     MJSW   = 0.5               )
```

This information includes the following key parameters:

- Device threshold voltage (`VTO`)
- Gate oxide thickness (`TOX`)
- Specific device capacitances (`C...`)
- Doping parameters (`NFS, NSUB`)
- Mobility (`U0, UCRIT`)

and several others.

In addition to the MOSFET modelling information, *Eldo* needs to know for each MOSFET the identity of the nets in the schematic to which it is connected, as well as the dimensions L and W of its channel, in order to evaluate the device voltages and currents. This information is extracted from the connectivity of the schematic, and by examining the properties set up for each device in *Pyxis Schematic*.

Finally, for realistic transient responses, *Eldo* needs to know the nature and magnitude of the parasitic capacitances (and sometimes the resistances) of the various interconnections. This information depends on the exact details of the layout, and is therefore not available directly from the schematic. In fact, it is obtained by an indirect process known as *Back Annotation*. In back annotation, parasitic elements in the layout are evaluated by an extraction program (*PEX* under *Calibre*) which takes into account the physical characteristics of the material involved (polysilicon, metal, etc) as well as the dimensions of the corresponding mask shape. The capacitance (and, where appropriate, the resistance) values are then inserted *back* into the corresponding schematic, appearing as *annotations* on the schematic display.

Mentor Graphics uses Design Viewpoints to handle the issue of back annotation. *A* designer may associate a number of different sets of extracted parameters - possibly corresponding to a number of different layouts under evaluation - with a named schematic, and may *connect* and *disconnect* these at will from the corresponding Viewpoint.

A typical complete *Eldo* input file is shown opposite, corresponding to an inverter, with comments. Note that this input data is not normally seen by the designer. Each basic element (*primitive*) is identified by a characteristic initial - for example, voltage sources by V, capacitors by C, MOSFETs by M, followed by a distinguishing identifier. This is then followed by a set of identifying numbers (*node numbers*) which indicate the way the standard terminals of the element are connected to other elements. The allocation of node numbers is fairly arbitrary, and bears no relation to any identifiers generated in *Pyxis Schematic*. For this reason there is a translation table which identifies the correspondences between Mentor entities and the Eldo node list. A series of directives beginning with a '.' specify the kind of analysis to be undertaken - for example, DC sweep, transient response, etc.

Note that information about the connections between the devices and about L and W comes from the *schematic*, and not from the *layout*. This may seem strange, since it is the layout that actually expresses the connectivity and device dimensions of the manufactured device, by defining the shapes and sizes of the corresponding mask shapes. This is a deliberate policy; the approach used by Mentor of *back-annotating* layout and parasitic information onto a schematic allows for the possibility of implementing the same schematic circuit in a number of different ways (for example, different layout styles, libraries or even different processes).

```
ELDO INPUT FILE  (INVERTER)
Title... SPICE netlist
.options  tnom=27
.temp=27
*
* Set up voltage sources for Vdd and A (DC supply and a pulse)
vdz2 501 0 DC 5
vdz1 503 0 PULSE 0 5 5e-09 5e-09 5e-09 1e-07 2e-07
*
* List nodal capacitances
cnn1 501 0 3.8499e-14
cnn3 503 0 3.48362e-14
cnn4 502 0 7.78117e-12
cnn5 504 0 3.4324e-14
*
* List MOSFET connections and models
*     D   G   S   B  terminals
miz2 501 503 502 501 CMOSP l=3.5e-07 w=1.5e-06
miz1 502 503 0 0 CMOSN l=3.5e-07 w=0.7e-06
*
* List MOSFET model information

.MODEL CMOSN NMOS (LEVEL=3 TOX=7.9E-9 NSUB=1E17 GAMMA=0.5827871 PHI=0.7
+VTO=0.5445549 DELTA=0 UO=436.256147 ETA=0 THETA=0.1749684 KP=2.055786E-4
+VMAX=8.309444E4 KAPPA=0.2574081 RSH=0.0559398 NFS=1E12 TPG=1 XJ=3E-7
+LD=3.162278E-11 WD=7.046724E-8 CGDO=2.82E-10 CGSO=2.82E-10 CGBO=1E-10
+CJ=1E-3 PB=0.9758533 MJ=0.3448504 CJSW=3.777852E-10 MJSW=0.3508721)

.MODEL CMOSP PMOS (LEVEL=3 TOX=7.9E-9 NSUB=1E17 GAMMA=0.4083894 PHI=0.7
+ VTO=-0.7140674 DELTA=0 UO=212.2319801 ETA=9.999762E-4 THETA=0.2020774
+ KP=6.733755E-5 VMAX=1.181551E5 KAPPA=1.5 RSH=30.0712458 NFS=1E12 TPG=-1
+ XJ=2E-7 LD=5.000001E-13 WD=1.249872E-7 CGDO=3.09E-10 CGSO=3.09E-10
+ CGBO=1E-10 CJ=1.419508E-3 PB=0.8152753 MJ=0.5 CJSW=4.813504E-10 MJSW=0.5)

* Define swept input source and transient analysis times
.dc vdz1 0 5 0.1
.tran 1e-09 2e-07 0
*
* beginning of translation table for device INSTANCE names
*  netlist_name              original_name
*
*  miz1                      /I$1
*  miz2                      /I$2
*  cnn1                      /N$208_cap_net
*  cnn2                      /N$6_cap_net
*  cnn3                      /A_cap_net
*  cnn4                      /Y_cap_net
*  vdz0                      d$0
*  vdz1                      d$1
*  vdz2                      d$2
* end of translation table for device INSTANCE names
*
*
* beginning of translation table for NODE names
*  netlist_name   original_name
*
*  0              //ground
*  501            /N$208
*  502            /Y
*  503            /A
* end of translation table for NODE names
.end
```

A typical *Eldo* input file (edited for clarity)

The implications of this approach are quite profound, however. How can we be certain that the layout we have designed, possibly quite independently of the schematic, actually matches the schematic in every applicable way? This is clearly something that must be checked as part of the verification process. One can visualise the chaos that would surely ensue if a design were fabricated on the basis of a schematic which simulated perfectly (using *ModelSim* or *Eldo*), but whose layout had a couple of missing interconnections or shorts that went undetected by eye.

Clearly, the correspondence between layout and schematic *must* be verified. In the Mentor Graphics package this is achieved by a procedure known as *nmLVS* (*nm Layout Versus Schematic*), in which the interconnection information and transistor details are taken from the layout and compared, feature by feature, with the corresponding elements in the schematic, and every discrepancy recorded.

The approach described is capable of taking into account as much information as can be obtained about parasitic elements inherent in the layout, and can lead to results of considerable accuracy. However, this accuracy is only achieved at a comparatively high cost in terms of computing overhead. It is generally reckoned that the computing time rises approximately as the square or cube of the number of devices present in the simulation. Hence, it is not normally possible to consider simulating with *Eldo* circuits consisting of more than a few dozen transistors. This means that most designs of any size must be broken down into manageable elements each of which can be modelled in isolation using *Eldo*. The detailed functional and timing results from *Eldo* can then be embedded in a higher level model (e.g. a VHDL model) which can be invoked in a high level simulation tool such as *ModelSim*, which is better suited to the needs of verifying entire designs.

This approach is well suited to the **semi-custom** style of design, in which a library of pre-characterised cells is made available to designers. The cells themselves are created (typically involving use of IC layout tools) and modelled by the library developer using simulators like *Eldo* and by direct measurements on fabricated designs. The resulting models are then supplied to the semi-custom designer, who may for most circumstances use these rather than carrying out detailed *Eldo* simulation runs. In certain cases, the designer will wish to use a combination of pre-characterised standard cells, plus custom cells designed for a specific purpose and modelled using *Eldo*. We shall use this approach during the course of this project.

Eldo operates interactively. A graphical presentation package *EZWave* provides the control interface to the *Eldo* simulation itself, and serves to display the results. For ease of interpretation, *Eldo* provides means for viewing the original schematic on which the simulation is being carried out, allowing *cross-probing*, an operation in which the designer selects nodes and nets in the original schematic in order to reference these in *Eldo* commands.

Eldo cannot operate directly on component schematics. It requires a design viewpoint to be created to provide it with the correct set of rules for interpretation of the schematic. In this project, the viewpoint needs to express the appropriate AMS conventions for design interpretation. Although *Pyxis Schematic* and certain other tools have the ability to create default design viewpoints, these do not contain the necessary references to the AMS models that we require. We therefore need to create appropriate design viewpoints for *Eldo* and *Pyxis Layout*.

Standard Project in VLSI Design SB1

Laboratory Guide 6 – Post-layout Verification and Simulation

This laboratory guide provides an introduction to some important techniques for analogue simulation of IC layouts using Mentor Graphics' *Calibre* and *Eldo* tools, which form a part of the ADMS package. For more details about *Eldo*, refer to the section: **Analogue Simulation & Eldo**.

The exercises in this session will use *Eldo* to give us detailed information about how the 2-input NOR gate **nor2x** (designed earlier in session 5 using *Pyxis* Layout) will operate when incorporated into the ring oscillator design. We shall first verify that the circuit behaves functionally as a NOR gate, with consistent input and output logic levels. Secondly, we shall determine (as accurately as possible) the rise and fall times of the gate, determined by transistor characteristics and parasitic capacitances. Our aim is to achieve fast but equalised rise and fall times at the output terminal of the gate, and we should be prepared for the possibility that modifications to the layout may be necessary to achieve this.

To put this session into perspective, let us first summarise the progress made thus far in the project. We have:

- developed the concept of the ring oscillator (RO)
- confirmed using VHDL modelling that the RO concept is valid
- explored the effect of varying NOR gate delays
- built a symbol and schematic for the RO based on the AMS library NOR20 gate
- predicted the timing characteristics of the RO based on the AMS NOR20 gate
- designed mask layouts for a 2-input NOR gate **nor2x**
- corrected any design rule violations in the **nor2x** layout

Note that although our own **nor2x** layout may be free of design rule violations, this is not sufficient to confirm that it actually represents a 2-input NOR gate! It is necessary to check that no errors have occurred in translating from the transistor schematic representation (perhaps via a stick diagram) to the layout. When we are satisfied of this, we need to confirm that the electrical performance is as required -- that is, that the particular combination of devices and interconnections actually functions as a NOR gate, and has the desired transfer and timing characteristics. Once these requirements are satisfied, the **nor2x** design may be incorporated in a library for use in the Ring Oscillator design or in other layouts where the NOR function is required.

Thus the key stages in the verification process are as follows:

- verify the correspondence between the **nor2x** layout and the **nor2x** schematic
- determine the parasitic elements present in the **nor2x** layout
- *back-annotate* the parasitics to the corresponding nets of the **nor2x** schematic
- simulate the result using *Eldo*
- account for parasitics arising when the **nor2x** gate is connected to other elements

Schedule

The remainder of this guide is divided into four key sections:

Section I The importance and assessment of parasitic capacitances
Section II Layout-Versus-Schematic (LVS) using *Calibre*
Section III Extraction of parasitic capacitances using *Calibre*
Section IV Simulation of the back-annotated **nor2x** schematic using *Eldo*

Note: section I contains an exercise to be carried out in advance of the lab session. Although it is not essential to do it before you carry out the exercise in section II on LVS, you do need to do it before you start section III or IV. You should try and organise your work so section I can be carried out away from the workstation between scheduled sessions.

Section I The importance and assessment of parasitic capacitances

The following section is best carried out away from the workstation.
You need a printed copy of your completed nor2x **cell layout in order to proceed.**

Appreciation of the parasitic capacitances associated with a design is vital. Delay will be introduced by *any* capacitive element that has to be charged and discharged while the gate is operating. This applies to many, though not all, of the capacitive elements found in your design.

Later we shall use the tools in the *Mentor* package to analyse these capacitances with considerable accuracy. However, this process cannot be completely automated. When considering a design it is very easy to overlook important contributions to capacitance and thus obtain an over-optimistic impression of performance. It is clearly important for a designer to have an appreciation of *which* contributions to capacitance have the most serious effect on performance, so that they can be minimised. The purpose of this section is to identify the most important contributions that need to be taken into account. Some are inherent in the layout for the **nor2x** cell (already carried out), while others will arise from interconnect used to link the cells. This interconnect has not yet been laid out, so its effects can only be estimated.

The following activity is to be carried out away from the workstation before proceeding with the automated extraction exercise. You will need a printed copy of your completed **nor2x** cell layout in order to proceed. If necessary, use the procedure in **Getting Started** and in Lab Guide 5 to produce one, or refer to the copy included as an Appendix to that Lab Guide. It may be useful when estimating dimensions to note that the size of the entire **nor2x** layout cell as supplied is 14.2 μm tall by 6 μm wide. You may wish to include one or more rulers in the checkplot to help estimate dimensions. Note also the availability of the *Pyxis* Layout command: **(Menu bar) Report > Selected**, which reports (among other things) the area in μm^2 of any layer currently selected.

Parasitic capacitances arising from interconnect

Several forms of interconnect are used in digital IC layout. These are basically **polysilicon** and **metal** layers (of which there are 4 in **AMS C35B4**). All are separated from the substrate by means of dielectric layers of SiO_2, polyimide or other material, and thus exhibit a defined capacitance to the substrate. The thickness of the dielectric is different in each of these cases, however, with the result that different materials have rather different specific capacitances (per unit area). Clearly there must also be

inter-layer capacitances – for example, wherever **poly1** is overlaid by **metal1, or metal1** by **metal2** – but we shall ignore these effects for this project.

Figure 1 below illustrates some of the capacitance contributions within a typical **nor2x** gate due to *interconnect*.

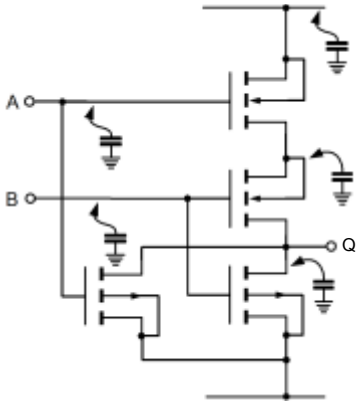


Figure 1: Some capacitances due to interconnect

Other contributions to parasitic capacitance

In addition to the parasitic capacitances arising from interconnect, there are contributions due to the semiconductor devices themselves. These can be identified:

- at each gate (where a MOS capacitor is formed with plates comprising the gate electrode and the channel; the dielectric is the thin gate oxide);
- at each drain and source, owing to the reverse-biased p-n junction formed between the source or drain and the substrate in which it lies.

These capacitances are closely controlled by the detailed dimensions of the devices, as well as by the characteristics of the materials used. We shall not evaluate the semiconductor capacitances here, as the models employed are quite complex. They will be determined later by *Eldo*, using extracted dimensions, and we shall consider the importance of these contributions at a later stage.

You can read about the evaluation of parasitic capacitance in CMOS gates in:
Microelectronic Circuits 5th Edition, by **Sedra & Smith**, chapter 10, section 10.2.3, pp 958-961, "Design and Performance Analysis of the CMOS inverter".

Analysis of parasitic capacitances due to interconnect

The next two sections will provide important data for inclusion in your Final Report.

The table below contains values provided by AMS. They comprise typical specific capacitance values for three main forms of interconnect. In order to apply these values, you must estimate the area corresponding to each element of interconnect. In fact this only gives us an approximation, because at these geometries the *sidewalls* of the interconnect add substantially to the capacitance, with a contribution tht depends on the *perimeter* of the material. It is also apparent that several shapes corresponding to different materials may be electrically connected. Strictly speaking, we should also evaluate the electrical resistance of each element of interconnect and consider the entire structure as a set of interconnected R-C ladders. To simplify matters we are going to assume that we can neglect the electrical resistance of the interconnect, and that capacitive contributions may be combined as parallel capacitances.

- Study the **nor2x** layout and schematic. Using data from the table, estimate the area capacitance (to within about 10%) between each interconnect mask shape and substrate, using reasonably accurate scale measurements of area. Annotate your layout diagram accordingly. NB 1fF = 10^{-15} F.
- Use the table templates in the Appendix to this Lab guide to present your data.
- Comment on the relative merits of the three forms of interconnect from the point of view of designing compact, high speed circuits.

AMS C35B4 Technology - Interconnect specific capacitances

Material	Polysilicon	Metal1	Metal2	Units
Specific capacitance to substrate	11.9×10^{-2}	3.2×10^{-2}	1.2×10^{-2}	fF μm^{-2} [1fF = 10^{-15} F]

Estimating delay in interconnected gates

In real logic circuits consisting of many gates, estimates of delays must take into account not only this delays due to the capacitances (and other parasitic elements) within any gate, but must also consider effects caused by any other logic gates connected to it, and by the interconnect used to do this. Clearly, there are many different gate types, and myriad ways of interconnecting gates, so the problem can easily become complex. However, a number of useful simplifications can be made.

When estimating the delay introduced by a logic gate in a circuit consisting of several gates, it is convenient to suppose that all the elemental capacitances (due to wiring, other gate inputs, etc) to which the output terminal is linked can be combined as a single 'lumped' capacitor. The time taken to charge/discharge this capacitance can then be determined by considering the conductances in the pull-up and pull-down chains.

Using this approach facilitates a straightforward approach to analysis of timing delay which, although approximate, is nonetheless realistic and gives sensible results.

We shall now assume that the output of your **nor2x** gate is to be connected using **metal1** interconnect to the **two** inputs of a succeeding identical **nor2x** gate (as in the ring oscillator). Both gates operate electrically as inverters - see Figure 2 opposite. The interconnect shown is indicative, and not specific to this design.

- Draw (by hand) a schematic representation (based on Figure 1 and Figure 2) of the situation that exists when two (or more gates) are connected in cascade. It should help you identify those capacitances that contribute to delay.

Note: It is very easy to overlook capacitive contributions and get an impossibly optimistic but meaningless estimate for the delay. Equally, it is important to be aware of those capacitances that are connected to static nodes (where the potential does not change). These do not affect delay, and including them would also give a false impression. You must take care to avoid falling into either of these traps!

- Making reasonable assumptions about the siting of the two abutted **nor2x** cells, estimate the area of **metal1**, assumed to be of width 0.9 μm, required to make these connections, and hence estimate its capacitance to substrate. You can deduce the form and size of the **nor2x** cell elements from the plot shown as an Appendix to **Lab Guide 5** (**IC layout with *Pyxis* Layout**). Then, being careful that you have considered *all* relevant contributions, estimate the **total** capacitance due to interconnect being driven by the first gate. Use the table templates given in the Appendix to help gather and present the data. Record the **total capacitance**, and annotate your paper schematic for later reference.

Delays in more complex logic systems

We are now in a position to account for the capacitive effects within individual logic gates themselves, and those arising from interconnect used to link gate inputs and outputs.

- Consider carefully which of the various capacitive elements will contribute directly to the delays observed in the case of a multi-stage design such as the ring oscillator. Study the transistor level schematics of Figure 1 and elsewhere, and the **nor2x** layout annotated with parasitics.

Use the table templates supplied in the Appendix to this Lab Guide to assemble and present these elements. Accordingly, update your schematic representing the cascaded gates of Figure 2. Annotate it to show all relevant contributions towards parasitic capacitance that affect delay. Some of these have been specifically mentioned above, while others have not, and you should think carefully about which contributions are likely to be of greatest significance in determining the speed of the gate. The table of specific capacitances (in Section II of this Guide, above) should be of help in deciding relative values.

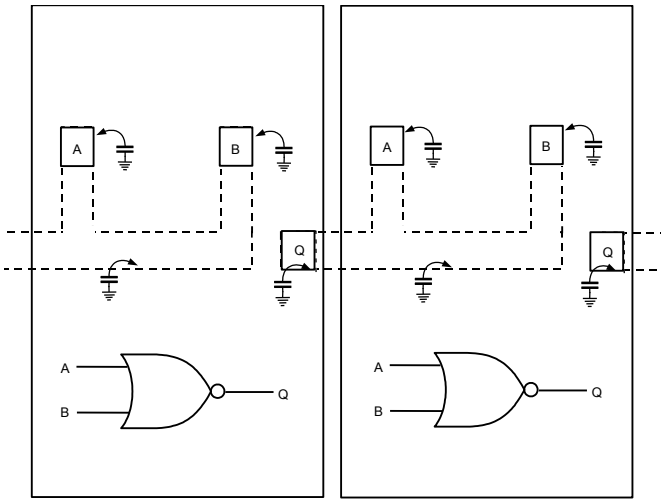


Figure 2: Estimation of additional interconnect capacitance

Section II Layout-versus-Schematic (LVS) using *Calibre*

In this section we'll check whether the layout you produced in Lab Session 5 is electrically equivalent to the schematic on which it is based. This will make use of the Mentor Graphics *Calibre* package, and you may need to conduct several runs through the package. Although there is a certain amount of setting-up to do first time around, *Calibre* stores most of its settings in **runset** files, so in later runs only changes in settings need to be entered.

You should be logged in at a workstation as described in the **Getting Started** pamphlet, with *Pyxis* Project Manager running. The current Mentor working directory should be **$CBT_WD** (which is the soft prefix for your **$MHOME/cbt** directory), and you should open up your layout library, created in Lab Session 5, step 1.

1. Check the cell

The **sb1_nor2_lib** came from another designer's account, in which the arrangement of directories and their relationship with the technology files in **$HK_C35** was different. In these circumstances, it is good practice to check the *references* associated with the components in the library to ensure they are intact, complete, and compatible with your project.

With *Pyxis* Project Manager open, navigate to the **sb1_nor2_lib** external library and highlight the **nor2x** cell within. Now give the command: **(Menu bar) > Edit > Check References**. The **Change/Fix** dialogue should open up. In the upper pane are listed the internal and external references of the nor2x design; all should have a tick near the left column. If not, it may signify structural differences between your project hierarchy and the sender's, or that parts have been missed out. If these are not corrected, problems will arise when the cells in the library are instantiated or otherwise involved in downstream processes. The lower pane provides a means for updating references to deal with such systematic differences. It cannot cope with elements that are missing, however.

Satisfy yourself that the **nor2** references are intact and if so, OK the dialogue box. If not, consult a demonstrator who will advise if any changes are needed.

2. Open the schematic

First open up *Pyxis* Schematic on the **nor2x** schematic. There is no special preparation required – this is the 'gold standard' against which the layout will be compared. To be specific, a 'netlist', derived from the **nor2x** layout, which summarises which component is connected to what interconnect, will be compared against a similar netlist derived from the **nor2x** schematic

3. Enter simulation mode to create a netlist

Enter simulation mode by clicking on the green **Play** button near the bottom of the left toolbar. A dialogue will pop up to ask for the configuration to use. For the moment, ignore the pre-existing configuration (this will have the keyword **SDL** embedded in it), and create a new one for *Eldo* and click **OK** to enter simulation.

4. Generate a netlist from the schematic

Select **(palette menu) Execute > Net list**. No other intervention is required, and Pyxis will generate an *Eldo* netlist.

By default the *Eldo* netlist will be placed at:

```
$CBT_WD/[Layout-Lib]/nor2x/[simulation configuration]/netlist.spi
```

Click on the button: **(palette menu) Results > ASCII Files > View Net List** to have the netlist presented in a *Notepad* window. It should be recognisable as a pared-down spice-like input file, which is one of the standard forms recognised by *Calibre*, the LVS checker. When ready, close the *Notepad* window.

5. Open *Pyxis* Layout on the nor2x layout

Revert to your *Pyxis* Project Manager window, which should be open on your Layout Library. Bring up a *Pyxis* Layout window by right-clicking on the icon corresponding to the **nor2x** layout, and selecting: **Open > Layout Editor**. When the *Pyxis* Layout window appears, enlarge it as far as possible.

6. Running *Calibre-Interactive LVS*

Invoke *Calibre LV*S by selecting **(Menu bar) Tools > Calibre > Run LVS**. In the ***Calibre-Interactive nmLVS*** window, click on the **Rules** button. The path to the rules file, which is the same as that for *Calibre DRC*, should be inserted automatically. Click on the **Input** button and check the settings indicated for each of the tabbed dialogues within (these should not need to be changed):

Run	: **Hierarchical**
Step	: **Layout vs Netlist**
Layout :	
Files	: **nor2x.calibre.gds** (in green, to be created)
Format	: **GDSII**
Export from layout viewer	: **checked**
Top Cell	: **nor2x**
Layout Net list	: **lay.net**
Netlist :	
Files	: **nor2x.calibre.src.net** (in red, already exists)
Format:	: **SPICE**
Export from schematic viewer	: **checked**
Top Cell	: **nor2x**

You can similarly look at the options available via the **Output** button, and via the **(Menu Bar) Setup > LVS Options** command, but don't change anything.

Click **Run LVS** to begin checking. A number of message panes will appear, reporting that *Pyxis* Schematic and *Pyxis* Layout are both exporting netlists to Calibre. After *LVS* has completed, *RVE* should be invoked automatically to view the results and diagnose any errors. If there are none, you should see the welcome smiley face. However, while (barring accidents in Lab Session 5) there should be no fatal discrepancies (e.g. missing connections or shorts), Calibre al;so checks the dimensions of key structures specified in the schematic (specifically L and W) for the MOS transistors and reports on any differences between them.

If you see messages regarding discrepancies, left-click the [+] signs for more information, and discuss them with the demonstrator. You must resolve them before moving on to the next section. You may need to make corrections to one of your designs.

III Extraction of parasitic capacitances using *Calibre-Interactive PEX*

This section needs to be undertaken at the workstation.

Before a proper simulation can be carried out it is necessary to determine the magnitudes of any parasitic components (for example, resistances or capacitances) inherent in the layout of the IC. *Calibre-Interactive PEX* provides facilities for extracting resistance and capacitance values in various forms, including lumped or distributed capacitances between pairs of layers as well as from each layer to substrate (ground). However, in this project we shall confine ourselves to determining only the lumped capacitances between interconnect and substrate (since these are a primary determinant of rise and fall time).

Important note: If you begin this section at the beginning of a session, you must first carry out an LVS check using the procedure described in **Section II** above.

7. Open *Pyxis* Layout on the nor2x layout

Open or restore the *Pyxis* Layout window, as necessary, and maximise it to fill the screen.

8. Running *Calibre-Interactive PEX*

Start *PEX* by selecting **(Menu Bar) > Tools > Calibre > Run PEX**.

First of all, click the **Rules** button in the In the ***Calibre-Interactive PEX*** interface. The **rules** entry needs to be corrected. Click on the triple dots button, and in the **Choose PEX rules file** dialogue, choose **_c35b4c3.rules_**. Click **OK**. At this point Calibre will make a local copy of the rules. Note that *Calibre* seems to require this

Now check the remaining settings in the *Calibre-Interactive PEX* tabbed dialogues. Note that some of the following will need changes.

Inputs
- Layout :
  - Files : **nor2x.calibre.db** (green)
  - Format : **GDSII**
  - Export from layout viewer : **checked**
  - Top Cell : **nor2x**
- Netlist :
  - Files : **nor2x.src.net** (red)
  - Format : **SPICE**
  - Export from schematic viewer : **checked**
  - Top Cell : **nor2x**

Outputs
- Extraction Mode : **xRC**
- Extraction Type : **Transistor Level | C+CC | No Inductance**
- Netlist :
  - Format : **DSPF** (this *will* need adjusting)
  - File name : **nor2x.pex.c+cc.dspf**
  - Use Names From : **SCHEMATIC**

Other options may remain at their defaults. Click **Run PEX** to perform extraction. A **PEX Netlist File** window with the extracted netlist appears when the extraction

has completed. Note that it contains two **.include** statements, which incorporate the results of Calibre's extraction operation. These are (in our case) parasitic capacitances; they can also include resistance, self-inductance and mutual inductance.

If necessary, click on the **RVE** button to open the *RVE* interactive results environment, and click on **Parasitics**. Study the values produced.

- Are the capacitances similar to the values you estimated by hand? **Comment.**
- Record the capacitances observed at the three main ports **A**, **B** and **Q**.
- Note also the capacitance values given for the **VDD!** and **GND!** nets. How do you expect these to affect the operation of your design?

9. Verify that *Calibre* responds to changes in layout

Refer to the section: **Analysis of parasitic capacitances due to interconnect** (above), which considers the capacitance due to interconnect used to connect the output of the gate under consideration to other gate inputs.

Using *Pyxis* Layout's commands, temporarily add to your layout a strip of **metal1** interconnect. Imagine that you plan to connect the output port **Q** to the input ports **A** and **B** of a fictitious gate, placed adjacent to the first, and lay out the additional interconnect accordingly. Refer to Figure 2 for ideas on how this might be laid out – only a reasonable approximation of the geometry is required. Do **not** save the modified layout. Re-run the *Calibre* PEX utility, and *observe and record* the modified capacitance values. Work out by how much the capacitance has increased by the addition of the metal interconnect piece.

Note that when calculating capacitance, *Calibre* takes into account all the contributions, arising both from the area, and from the perimeter of the capacitor plates (i.e. including the sidewalls of the interconnect). The result is likely to be quite different from the values you estimated by hand based on area alone.

Exit from *Pyxis* Layout, first closing *Calibre*'s windows. Save the *Calibre* runset files, to speed up future runs, but when prompted to save changes to the layout cell, it will be best to uncheck the **nor2x** box, so that the temporary interconnect is not saved.

IV Simulation of the schematic design using *Eldo*

In this section you will run the *Eldo* simulator to determine the electrical behaviour of your **nor2x** design when driven with typical signals. The results you obtain will give you a good indication of the efficiency of your design, and should show evenly matched rise and fall times of a few nanoseconds, dependent upon the details of your layout.

Setting up for *Eldo* consists of three broadly separate operations:

- Creating a test-bench schematic in *Pyxis* Schematic, containing ports, supplies and an instance of the **nor2x** symbol; checking and saving; then, in Simulation Mode,
- preparing power supply forces, signal forces and any other stimuli;
- running the simulation and inspecting the results.

You have already undertaken similar activities in preparing for simulations with library cells in Labs 3 and 4. Working at transistor-level is very similar, though some of the details are different.

Before you proceed with the *Eldo* section, you should have completed all previous sections in this guide, and you should have a *Pyxis* Schematic session running.

10. Create a test-bench schematic

Open *Pyxis* Schematic and create a new test-bench schematic – you are free to choose a name for this yourself, but something based on **nor2x-testbench** would be appropriate. Take care not to clash with your partner's choice if you decide to do this work independently.

This needs to contain an instance of the **nor2x** symbol, together with appropriate ports, power references and wired connections. Use the method of steps 12-15 in Lab Session 4, making sure to check and save the result.

11. Enter simulation mode

With *Pyxis* Schematic open on the test-bench schematic, enter simulation mode by clicking on the green **Play** button at the bottom of the left toolbar. A dialogue will pop up to ask for the configuration to use. Select a new standard *Eldo* configuration, and click **OK** to enter simulation mode.

12. Applying back-annotation to the schematic

Click on the **nor2x** instance in the Simulation schematic window, and select the command: **(Menu bar) > Parasitics > DSPF**. Enter the following details in the DSPF pop-up window.

Choose DSPF	:	**nor2x.pex.c+cc.dspf**
Level	:	**RC**

Note you will need to use the Navigator to find the **dspf** file. Its default site, determined by the *Calibre* run in step 8, is in:

```
$CBT_WD/[Layout-Lib]/nor2x/nor2x.cal
```

Click **OK** to back-annotate the parasitics to your design. The **nor2x** instance should now be annotated in red with the **dspf** file name and the annotation level. In the event that you wish to remove the parasitics for whatever reason, go to **(Menu bar) > Parasitics > Remove DSPF.** You may freely connect and disconnect back-annotation items, so some care is necessary to make sure any later simulation runs with the intended ones.

The individual parasitics can now be explored by opening down into the **nor2x** symbol. If this is not already highlighted, click on it to highlight it, then give the command: **(Menu bar) File > Open Down**. The screen will change to display the transistor-level schematic **nor2**.

Select any net for which you wish to see the parasitic – for example, **Q**, **A** or **B**, and with the right mouse button, select the command: **Parasitics > Show Net Parasitics**. A panel will be displayed enumerating all calculated parasitics for the net or nets selected. Scroll down the panel to observe the information. If you give the command: **Parasitics > Show Lumped Capacitance** this will show the

accumulated sum capacitances to substrate and capacitances to other conductors, normally in the parent cell – give the command **(Menu bar) > File > Open Up** to return temporarily to see this.

13. Set up Eldo to perform a DC Sweep analysis on the nor2x gate

We shall first determine the transfer function of the gate with input **A** fixed (0 volts), while varying input **B** over the range 0 – 3.3 volts. Before the simulation can begin, we must first set up some *Eldo* forces to determine the required voltage and signal sources. In most cases this can be achieved by selecting nets in the schematic, but some typing will also be required.

First we set up those nets we wish to monitor, by selecting them, then using the (**Palette menu**) > **Outputs** command. Add the required nets to the **Objects** pane, making sure the required task is set to **Plot**. Close the **Setup Simulation** panel. Now set up the power supply. As in Lab 4, a DC supply of 3.3V is required (for the moment) on **VDD!**, and a 0V supply for **GND!** (both relative to **GROUND**). Use the (**palette menu) > Forces/ICs** command to do this; add these forces and close the **Setup Simulation** box once again.

Now we set up a static voltage of zero at each of the inputs **A** and **B**, referenced to **GROUND**. We shall later specify that **B** will be varied in order to determine a transfer characteristic. Select these two only, and click on **(Palette Menu) > Forces/ICs > Add**. Select the DC tab and enter **0V** as the value. Add both nets to the Objects pane, and close the **Setup Simulation** box once again.

Check your set of monitors and forces is complete before continuing.

14. Run simulation to determine the transfer characteristic of nor2x

Now set up the simulation run. Click the **(palette menu) > Analysis** button. In the resulting dialogue box, check **DC** and **Enable DC.** Select Source (Force) in the **Sweep Type** box, and click on the Selection button to the right. Choose **VFORCE__B** as the source to be swept. In the **Start**, **Stop** and **Step** fields enter **0V**, **3.3V**, and **0.3V** respectively. Click **Apply** and close the **Setup Simulation** dialogue box.

To run the simulation, click the **Play** button in the left palette menu. If all is well, the simulation should run within a second or two. Click the **EZWave** button in the left palette to call up the waves window when simulation has completed. Note the availability of screen cursors to help with quantitative measurement. It is also possible to print the active graph using the command provided in the **Getting Started** pamphlet. Please ask if you need further details for setting up printing.

- From the graphs, determine the maximum linear amplification of the gate with respect to input **B**, i.e. ($\left|\partial V_Q / \partial V_B\right|$).
- At what input voltage level is maximum linear gain achieved?
- Consider the accuracy of the measured linear gain value. Could you set up the simulation conditions (step 14) in a slightly different way in order to enhance the accuracy obtained?
- Determine the input switching level (at which $V_Q = V_B$).
- Now determine the **transfer function**, **gain** and **switching level** relative to input **A**.

- Estimate the **noise margins** (3B2 course notes, or see Sedra & Smith or Weste & Eshraghian).
- How might you use *Eldo* to determine the transfer function with **both** inputs being swept simultaneously (rather than just **A**)?

From the foregoing, you should be in a position to decide whether or not the circuit has suitable transfer characteristics for use as a NOR gate. However, this analysis has given no information about the *dynamic performance* of the gate (i.e. its speed), which depends greatly on the parasitic components evaluated by *Calibre*. The following paragraphs detail how to carry out a transient response analysis of the **nor2x** design.

15. Set up a Pulse stimulus for use in transient analysis

In order to predict the rise and fall time of the **nor2x** gate, and hence the delay imposed by it on digital signals, we must carry out a transient analysis. This is relatively straight-forward in *Eldo*, where a transient response is constructed by evaluating nodal voltages and currents at intervals determined by a *time-step* until a specified *stop-time* is reached. We also have to specify at least one input signal as a time-varying function. *Eldo* offers several possibilities for this, including sinusoids, pulses, exponentials, or piece-wise linear waveforms. In digital circuit modelling, the most useful of these is the pulse waveform. To specify a pulse, we need to specify a number of parameters. You have used this approach already, but for completeness, these are illustrated by the waveform in Figure 3 below.

It is important to establish realistic values for the **Rise** and **Fall** times. They should be about the same as the rise and fall times of typical gate outputs; for initial experiments, the suggested settings of 0.1 ns should be satisfactory.

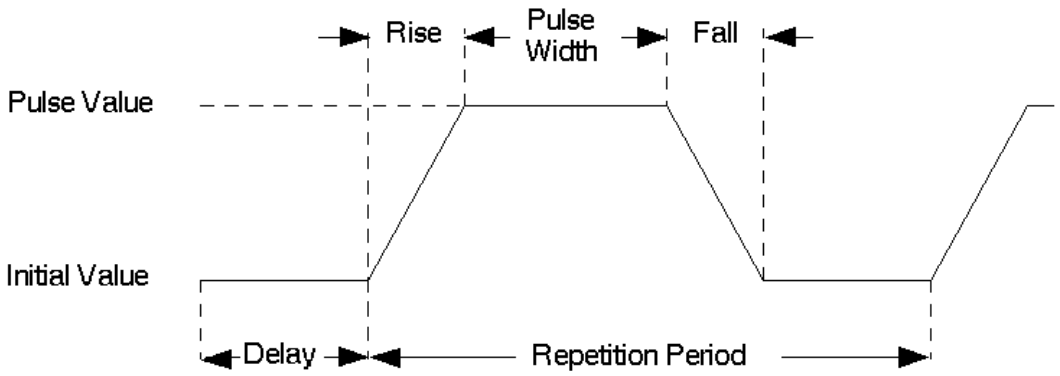


Figure 3: Force Pulse Signal Parts

To set up the analysis, click the **Analysis** button in the palette menu. When the **Setup Simulation** dialogue appears, check the **TRAN** checkbox. Note that it is possible here to adjust the **Stop Time** and the **Time Step**; 100ns and 1ns respectively will be suitable for the moment. Make sure that **Print Interval Time** and the **Maximum Step Time** are set to about 1% of the **Stop Time.** Click **Apply**, and close the dialogue box.

Note that an alternative way to achieve the same waveform is to use the **PWL** 'Piece-Wise Linear' option.

For the time being we shall keep the signal connected to input **A** fixed at 0 volts, and apply a pulse waveform to input **B**. Verify that the signals applied at **VDD** and **A** are still correct by viewing the summary in the white circle on the test-

bench schematics; if necessary, call up the dialogue to change the force by clicking on the white circle; correct or upodate as appropriate. To apply a pulse waveform to input **B**, double-click on its white circle. When all values are appropriately set, click **Apply** and close the dialogue box. Reconfirm the waveforms of interest again for observation using the same method described in Step 13.

16. Carry out a transient analysis on the nor2x gate

Use the **Play** icon in the left palette to have *Eldo* to initiate the simulation.

Once the simulation is complete, call up the waveform window. Again, you may wish to use cursors to help determine accurate rise/fall/delay times, or to generate a print-out. **When recording your results you *must* state the capacitance being driven, as without this information your results are meaningless**.

- Determine the output rise/fall time (10% to 90%) under these conditions.
- Determine the delay between input stimulus and output response, by measuring the delay that elapses between the input signal passing through VDD/2 and the output signal passing through the same level.
- Determine a suitable **PULSE** force to apply to input **A** so that the combination of signals applied to **A** and **B** exercises the **nor2x** design through all possible input and output states. Compare the rise/fall delay times for every possible input combination.
- If time permits, investigate the effect of varying the rise time/fall time of the applied pulse waveform(s).
- Use the table templates in the Appendix to help with presentation of the data.

17. Determining the effect of external parasitic elements

In an earlier section you evaluated 'by hand' the capacitance associated with the metal interconnect required to connect the output of the **nor2x** cell to the inputs of an adjacent **nor2x** stage (as in the ring oscillator), and thus determined the total capacitive load driven by the original gate. However, the transient response you just determined was for the **nor2x** cell totally isolated from any other circuits. Under these conditions, which are not truly realistic, *Eldo* cannot be expected to give accurate values for the expected ring oscillator frequency.

One way to circumvent this would be to edit the **nor2x** layout temporarily, adding metal and other shapes typical of the kind of interconnect we might reasonably expect. *Calibre* could then be used to re-generate more accurate back-annotations. Alternatively, an additional *ideal capacitor*/s may be added to the schematic representation of the cell from the library, with values chosen to represent these unaccounted-for contributions.

If you wish to try this approach, thje demonstrators will be pleased to help you: if you do so, record your results and compare these with your hand calculations. If you do not, do be aware that your simulation results so far will be overly optimistic because these elements have been ignored. However, it is possible with a little more work to carry out a more complete *Eldo* simulation run on a back-annotated top-level representation of part of the ring oscillator; and with rather more work, to do a full *Eldo* simulation of the entire design, taking full account of wiring parasitics.

18. Investigate the effect of varying supply voltage VDD

The effect of the supply voltage VDD in determining the conductance of the transistor channel was discussed in an introductory sheet, and you may by now have had the opportunity to investigate the measured performance of a real ring oscillator at various different supply voltages. We are going to use *Eldo* to model the performance of our ring oscillator design at a reduced supply voltage of 2 V, and hence predict the dependence of oscillation frequency on VDD. In order to investigate this using *Eldo*, it is necessary to modify the specification for the voltage source set up in Step 1, as well as the amplitude of the pulse stimulus set up subsequently for the transient analysis.

On the schematic, find the appropriate net corresponding to the VDD supply rail, and double-click on its white circle to modify the supply voltage VDD to 2 V. If necessary, change the magnitude of the signal applied to input **A** so it lies within the range 0-2 V. Finally, double-click on the white circle of net **B** and change the amplitude of the pulse signal applied to 2 V. Use the **(Palette Menu) > Forces** command or study the white-circle annotations to confirm these modifications have been correctly carried out. Carry out a transient analysis as in Steps 15-16, and compare the results obtained. Compare these also with the results obtained from direct measurement on the experimental ring oscillator (bearing in mind the enormous differences the processes used – dimensions differ by a factor of about 6.

19. Investigate the effect of varying W/L

The effect of aspect ratio (W/L) on worst-case gate delays has already been described. Careful choice of W/L is required to ensure that the delays corresponding to rising and falling signals are matched. You should by now have discovered how effective this matching process has been in your own case! In any event, we shall now briefly investigate the effect of modifying the W/L ratio for the p-channel transistors. This could clearly be achieved by modifying the original design (the schematic in this case, rather than the layout); however, it is more convenient to carry out this experiment directly in *Eldo*. This can easily be achieved, because the information is actually transmitted to *Eldo* by means of the **Edit Object Properties** dialogue.

With the schematic window active, **open down** into the **nor2x** transistor-level schematics and click to select one of the two **p-channel** transistors. Give the command: **(Menu bar) > Edit > Edit Object**. When the dialogue box appears, a list of variables is available for editing. Change W in such a way as to alter the aspect ratio by a factor 2-3. Repeat this procedure for the remaining p-channel transistors. Re-run the simulation and determine, and record, the new rise and fall times. If time permits, collect sufficient results to allow you to plot a graph showing the variation of delay time with W/L. **Note**: it is also possible to perform a sweep to determine performance as a function of (say), W.

20. Collect and present the results from this session.

Gather the accumulated simulation results together and present them in an appropriate form using tables or graphs – a number of suggested templates are given in the Appendix. You should include as many as possible of the following conditions:

- DC Sweep Analysis, giving data about the gate's transfer function (Step 14)

- transient response with internal circuit parasitics (Steps 15-16)
- transient response with internal/external parasitics (Step 17)
- transient response incorporating other effects (Steps 18 - 19)

Be sure to state **very clearly** the conditions applying to the simulations. For example, in transient responses, the results are meaningless if the aggregate driven capacitance is not quoted. It is very easy to ignore capacitive effects and get an impossibly optimistic figure for the delay. You must take care to avoid falling into this trap.

Include also in your tabulated results any relevant results obtained from measurement on the experimental ring oscillator.

21. Use *Eldo* results to model the behaviour of your design with VHDL

Select those simulation results you consider to be the most realistic and characteristic for your **nor2x** when used in the ring oscillator design (explaining why). Then use the procedure of Laboratory Guide 2 (**Functional simulation with vsim**) to adapt a behavioural model and investigate the overall system behaviour of your ring oscillator design based on the predicted timing characteristics of your **nor2x** cell. Hence determine the key timing parameters of the proposed ring oscillator module designed in this way.

Conclusion

The last few steps should have given you some insight into the issues that most directly affect the behaviour of fast CMOS logic gates. We shall leave the verification process at this point, but should perhaps observe that there is considerably more that could be done.

- Although we have made reasonable estimates of interconnect capacitances, the real **ring_oscillator** has not yet been laid out, so we do not know what parasitic capacitances might arise owing to the interconnect used to link outputs to inputs, etc. A slightly different kind of extraction can elicit this data once the layout has been performed. We will not pursue this here.
- *Eldo* is also suitable for modelling the entire design. At this stage, having assured ourselves of the satisfactory performance of our **nor2x** cell, we will continue to use *Eldo* for the remaining simulation work at higher hierarchical levels.

The remainder of the project is concerned with the use of semi-custom tools associated with *IC Station*, to prepare a set of masks for the entire **top_level** design, ready for despatch to the fabrication facility.

Laboratory Guide 6 – Appendix I

The following suggested table templates may be of help in the presentation of material in the sections of the Final Report relating to this Lab Guide. Please adapt the formats to meet your own specific requirements.

Interconnect capacitances

Index	Interconnect type	Description	Area / μm^2	Capacitance / fF
1				
2				
3				
….				

A similar table may be used to present calculated or extracted capacitance values.

Capacitances associated with ports

This table may help with assessment of the parasitic capacitance being driven by each gate output owing to coupled interconnect and input ports.

Port Identifier	Area references	Capacitances / fF	Total Capacitance / fF

A similar table may be used to present calculated or extracted capacitance values.

Areas and perimeters of transistors

Transistor ref.	Source		Drain		Gate	
	Area (μm^2)	Perim (μm)	Area (μm^2)	Perim (μm)	Area (μm^2)	Perim (μm)

Rise, Fall and Delay times for transient inputs

Input conditions	O/P Rise / Fall	Original setup		External capacitances included		Reduced supply voltage		Doubled W/L ratio for p		Other conditions	
		Rise/fall ns	Delay ns	Rise/fall nsl	Delay ns	Rise/fall ns	Delay ns	Rise/fall ns	Delay ns	Rise/fall ns	Delay ns
A falls	Rise										
B falls											
Both fall											
…..											

Similar tables to those above may be used to advantage in other parts of the report.

Standard Project in VLSI Design SB1

Laboratory Guide 7 – Top-level Design and Simulation

This Lab Guide will be issued later in the project

Standard Project in VLSI Design SB1

Laboratory Guide 8 – Top-level Layout and Integration

This Lab Guide will be issued later in the project

Standard Project on VLSI Design SB1

Electrical Characterisation of CMOS Ring Oscillator

This pamphlet describes a laboratory activity based on an integrated circuit originally designed and tested as a student project. Its purpose is the measurement of the switching speed of some CMOS logic gates on a 2 μm n-well technology silicon integrated circuit. The MOSFETs employed have threshold voltages of ±1 volt approximately. Although this is quite different from the parameters that apply to the AMS technology used currently in the project, the principles are the same. You will have an opportunity to investigate the behaviour of these circuits as a function of supply voltage and in a range of configurations, and compare with simulated results.

1. Test chip layout

As you can see from the optical micrograph in Figure 1 and the corresponding diagram in Figure 2(a), the upper and lower ring oscillators contain 113 and 115 gates of the same type, respectively. The input and output pads to the circuits are as shown in Table 1; the inputs have multiple functions as noted below. INPUT and OUTPUT numbers are those numbers assigned on the test board with the jumper wires, and the pin numbers are the pin assignments on the chip. A circuit schematic of the test unit is shown in Figure 3.

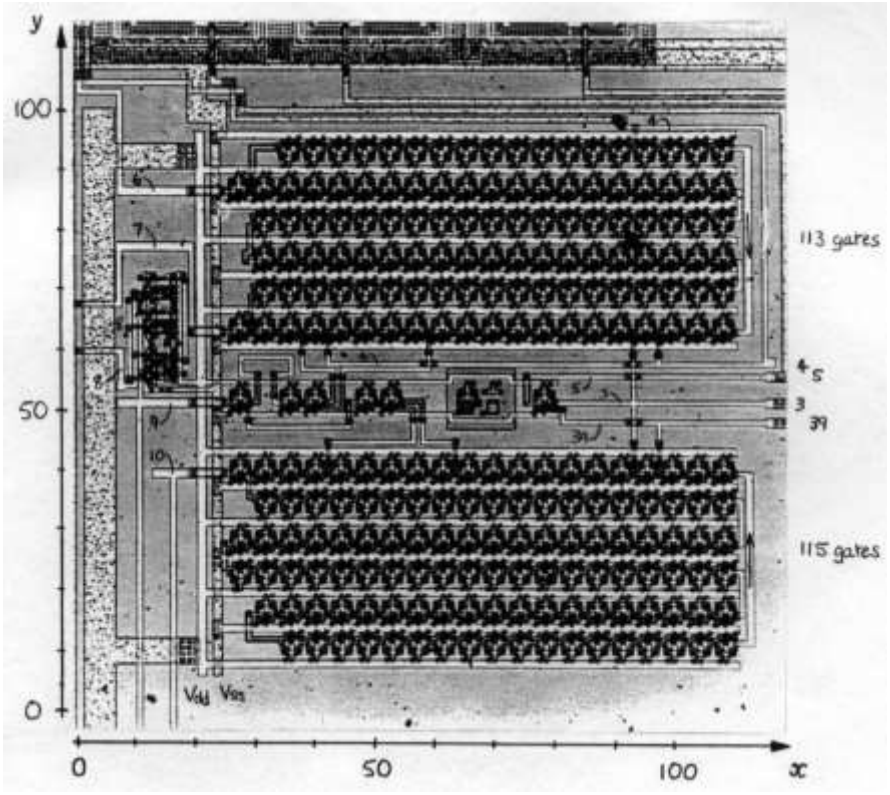


Figure 1 Photomicrograph of part of ring oscillator IC

The power supply V_{dd} and ground V_{SS} are supplied on the 2nd and 1st levels of metal on lines x=18 and x=21, pins 32 and 12 respectively.

The 6 logic gates in the area between the oscillators (grid reference x=85, y=53 and x=28-55, y=53) are similar to the elements in the ring.

There is an edge triggered divide-by-two circuit at (x=10, y=65) in Fig.2.

Input	Description	Grid Ref
INPUT 3 (pin 3)	Controls element 1 of ring-113 Controls element 1 of ring-115 Controls one input to the single device	(x=100, y= 66) (x=108, y=400) (x= 85, y= 53)
INPUT 4 (pin 4)	Controls three elements of ring-113	(x= 40, y= 66) (x=108, y= 66) (x=108, y= 97)
INPUT 2 (pin 39)	Controls element 115 of ring-115 Controls one input to the single device	(x=115, y= 40) (x= 85, y= 53)
OUTPUT 5 (pin 5)	From the single device	(x= 85, y= 53)
OUTPUT 6 (pin 6)	From the ring-113, after an extra buffer element	(x= 30, y= 91)
OUTPUT 7 (pin 7)	From the ring-113, after an extra buffer element	(x= 30, y= 65)
OUTPUT 8 (pin 8)	From the divide-by-two circuit which has output 9 as its input	(x= 10, y= 65)
OUTPUT 9 (pin 9)	From the fifth logic gate in the line (no buffer element)	(x= 26, y= 53)
OUTPUT 10 (pin 10)	From the ring-115 after element 16 after an extra buffer element	(x= 30, y= 40)

Table 1 Input and output pins

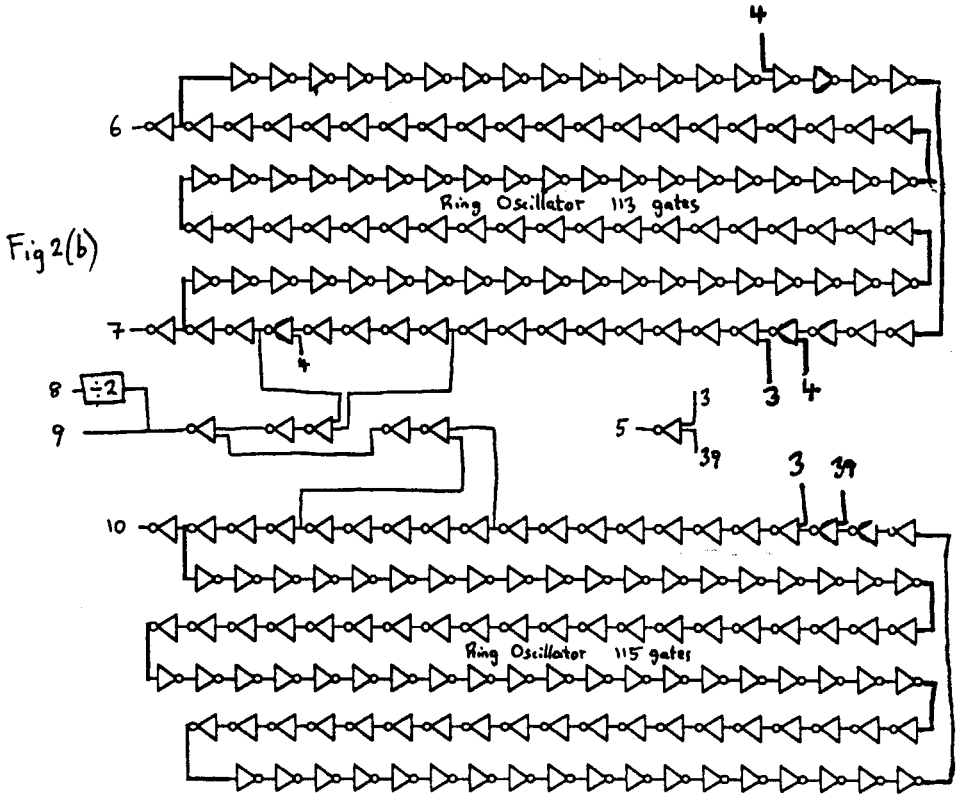


Figure 2 Ring Oscillator Circuit Schematic

2. Experimental Procedure: Testing the single device.

Important: Be sure to use Box A for sections 2-8.

Before measuring the ring oscillator we shall first test the single gate (x=85, y=53) on which both rings are based. Ensure that the measurement box in Fig. 3 contains a chip with design 21.

To change chips, move the lever over to release the pins, lift out the chip (without touching the pins - this could damage the circuit by electrostatic discharge). Push the pins into the pad of protective black conductive foam for storage, mount chip with dots aligned, and pull the lever to clamp the pins.

The supply voltage can be set to any value in the range 0.2 – 9 V approximately using the control knob at lower right on the test box. The meters provided allow accurate measurement of supply voltage and current. Note from Fig. 3 that current consumption is sensed by measuring the voltage drop across a 1Ω resistor – the meter must be set to measure voltage and the result converted mentally to a current. Setting the meter to a current range will give incorrect results.

- Please record the serial number of your chips: (e.g. 21-10) .
- Ensure that the positive power supply is connected to pin 32 on the chip and the ground is connected to pin 12. Set the power supply V_{DD} to 3V.
- Set INPUT 3 (pin 3) HIGH.

The input and output pads are **inverting**; i.e. the three-way switch in the **left** position (labelled HI) to one of the chip inputs gives a LOW input at the internal circuit. **Note**: the switches are not debounced and can introduce multiple pulses into the ring.

- Set INPUT 4 (pin 4) LOW
- Feed a square wave to INPUT 2 (pin 39) (switch set to the right - BNC input), amplitude about 3V, frequency about 1MHz
- Sketch what you observe when OUTPUT 5 and INPUT 2 (pin 39) are displayed together on a dual trace oscilloscope.
- Write down the logic function performed by the single gate.
- Use the oscilloscope to estimate the gate delay. What factors limit the accuracy you can expect to achieve?

3. Testing ring-115.

Ring oscillators **ring-113** and **ring-115** are made up of 2-input gates identical to that examined in section 2; **ring-115** has two inputs which can be used to control the oscillation. First examine the behaviour of the ring when free-running with V_{DD} at 3V.

- Set INPUT 2 (pin 39) to HI to allow the ring to free-run. Trigger the oscilloscope with OUTPUT 10 and observe the waveform.

Caution: The ring may go into a high order resonance owing to switch contact bounce. If so, first set INPUT 3 to LO to stop the ring and then set it to HI again. You may need to repeat this procedure more than once. Alternatively, with the switches set as required, reduce the supply voltage to zero and bring it up gradually to the desired value. This normally achieves a smooth start to the oscillations. **You are strongly recommended to monitor all output waveforms** using the oscilloscope, to guard against inadvertently selecting a harmonic mode.

Now investigate how the oscillation can be gated on and off by means of a pulse train.

- With the square wave connected to INPUT 2 (pin 39), adjust the input frequency at INPUT 2 (pin 39) to about 100kHz with a duty cycle of 50%.
- With the switch at INPUT 2 (pin 39) set to INPUT, observe OUTPUT 10 and INPUT 2 (pin 39) on the oscilloscope, using the input as a trigger. Sketch the waveforms.

Show how to use this approach to determine the ring frequency as a multiple of the pulse frequency, and record the ring frequency obtained in this way.

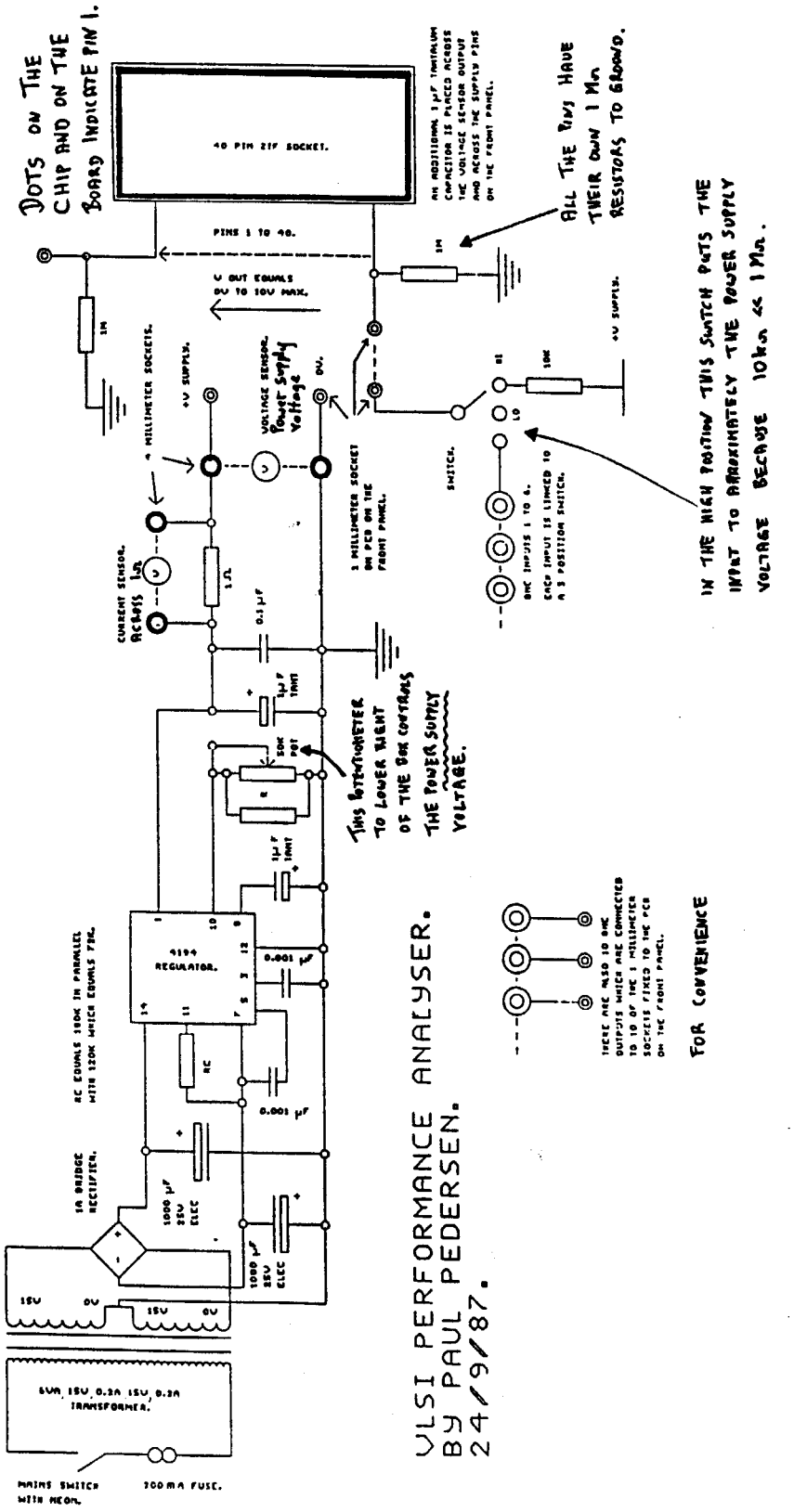

DOTS ON THE CHIP AND ON THE BOARD INDICATE PIN 1.
40 PIN ZIF SOCKET.
PINS 1 TO 40.
ALL THE I/P'S HAVE THEIR OWN 1 Mn RESISTORS TO GROUND.
IN THE HIGH POSITION THIS SWITCH PUTS THE INPUT TO APPROXIMATELY THE POWER SUPPLY VOLTAGE BECAUSE 10kn << 1 Mn.
THIS POTENTIOMETER TO LOWER SLIGHT OF THE BOX CONTROLS THE POWER SUPPLY VOLTAGE.
FOR CONVENIENCE
VLSI PERFORMANCE ANALYSER.
BY PAUL PEDERSEN.
24/9/87.
4194 REGULATOR.
MAINS SWITCH WITH NEON.
100mA FUSE.

Figure 3 Test box circuit schematic

4. Determining the gate delay by measurements on ring-115 and ring-113.

With the setup of section 3, carry out the following experiments:-

- Make a rough measurement of the period of the high-frequency oscillations from the oscilloscope.
- Using the most suitable method to determine the natural ring oscillator frequency (counter-timer or oscilloscope), find the gate delay per stage in the ring with V_{DD} set to 3V, making sure that you are not in error by a factor of 2!

- Now using **ring-113**, investigate a further way to estimate the gate delay. Display OUTPUTS 7 and 6 on the dual trace oscilloscope, set **ring-113** oscillating and divide the observed delay by the number of intervening gates.
- Find the number of gates by counting the gates in Figure 2, bearing in mind that the buffer gates on each output are external to the ring.
- Does OUTPUT 7 come before OUTPUT 6, or *vice versa*?

Compare and contrast the methods explored in sections 2 - 4 for obtaining the gate delay.

5. Measure the effect of varying the power supply voltage.

The switching characteristics are expected to depend on the supply voltage V_{DD}. In this section you will make a detailed investigation of the nature of the variation over a wide range of voltages.

- First determine the minimum power supply voltage required to establish full-amplitude oscillations by setting the switches so that only one ring free-runs, the other ring being off. Determine also the lowest frequency at which oscillations can be detected. Is this consistent with what you know of the characteristics of the transistors?
- Measure the frequency of the free-running ring using the frequency counter as the power supply voltage is varied. Plot the result and note the rate of change in frequency with power supply voltage V_{DD} **at 3 V**.
- Interpret your plot of frequency vs. ring voltage. Is it an accurate straight line? Suggest reasons for any deviations.

6. Performance comparison with different transistors in the ring

As explained, test chip designs 28 and 21 have similar transistors in the elements of the ring oscillators except that the p-channel transistors in design 28 are three times wider than those in design 21. This affects both the **on**-state channel conductance *and* the capacitive load presented to the previous stage of the ring.

> *To change chips, move the lever over to release the pins, lift out the chip (without touching the pins as this could damage the circuit by electrostatic discharge). Push the pins into the pad of protective black conductive foam for storage, mount the new chip with the dots aligned, and pull the lever to clamp the pins.*

What is the percentage difference in performance between the designs at supply voltage V_{DD} = 3 V and V_{DD} = 5 V? Explain on the basis of the transistor dimensions why you would expect a difference, and estimate the magnitude of the expected change. Make reasonable assumptions (with explanations) about parameters not explicitly stated.

7. Simulation of the device performance.

The output of an *Eldo* simulation of a portion of the ring oscillator is given in Figure 4, which also contains a sketch of how the five gates are interconnected in the simulation.

The voltage waveform V(A) consisting of a linear ramp up from 0 to 5 V, a flat region and a linear ramp back down to zero, is supplied to the input to gate A. The outputs of gates B, C, D and E are determined using *Eldo*. Note that the simulation provided corresponds to chip design 28 rather than the design just measured, and assumed V_{DD} = 5 V. Test chip designs 28 and 21 have similar transistors in the elements of the ring oscillators *except* that the p-channel transistor in design 28 is three times wider than that in design 21. This affects both the **on**-state channel conductance *and* the capacitive load presented to the previous stage of the ring.

Highlight in a bright colour the waveforms C and E on the diagram. Explain how waveforms B-E arise, and why the shape of waveforms C and E are similar to each other. Why are they different in detail from waveform A?

Using the curves in Figure 4, estimate the delay in the ring oscillator and compare it with your experimental measurement for 5V power supply voltage.

It may also be helpful to know that the *Eldo* run carried out takes into account the MOSFET channel resistances and all parasitic capacitances to substrate, but does not model other resistances in the circuit, such as the 2 μm wide polysilicon lines interconnecting the devices.

8. Stroboscopic pulse generator

This experiment requires a little extra determination, but carefully carried out, gives deep insight into the subtleties of the performance of the circuit, and of the accuracy of the simulation results that guided its design. Small changes in supply voltage can have a dramatic effect on your observations, owing to the strong dependence of gate delay on V_{DD}. Also, chip designs 21 and 28 have slightly different characteristics and it may sometimes be easier to observe the effects on chip design 28.

In the schematic of Figure 4, the sense gate F with inputs from E and B gives a high output only when both inputs are low. This function is also realised on the chip by the gates at (x=40, y=53) and at (x =55, y=53). The result appears at OUTPUT 9 (pin 9). There is also an edge-triggered divide-by-2 circuit on the chip at (x=15, y=60). Its input is internally connected to OUTPUT 9 and can be monitored at OUTPUT 8 (pin 8).

Given that the sense gates on the chip each span 5 elements of the rings (note that in Figure 4 only 3 are spanned),

- Describe the circumstances in which the output from the sense gate will be in the logic HIGH state during normal operation of the ring.
- What output do you expect from OUTPUT 9 of chip design 28 in Figure 2 when:-
  a) **ring-113** and **ring-115** are running freely with the same V_{DD} – consider what their relative frequencies are expected to be;
  b) when INPUT 4 is LOW (i.e. HIGH after inversion at the input pad), and **ring-115** alone is running?
- Verify your predictions by experiment. **Change to chip design 28 only if necessary.**
- First, observe OUTPUTS 10 and 9 on the oscilloscope with both rings free-running. To achieve this, switches at INPUTS 2-4 must be set to HI. Use a low supply voltage of approximately 0.6 - 0.7 V, and guard against either ring entering a harmonic mode. Once you have observed the sense waveform, consider applying it to the oscilloscope's External Trigger input so you can monitor both ring outputs in the vicinity of the sense signal.

- Confirm that the circuit performs the expected function over a small range of supply voltages and increase/decrease the power supply voltage until it just **no longer** works. Record your observations.
- Next, observe OUTPUTS 10 and 9 on the oscilloscope with a fairly low power supply voltage, and **ring-115** only in operation. To achieve this requires the switches at INPUTS 4, 3 and 2 (pin 39), set LO, HI and HI respectively. Investigate the divide-by-2 output, at OUTPUT 8 (pin 8), and show that it works as expected over a range of supply voltages, and record the range. Outside this range the rise/fall of the input edge may be too slow or too fast to clock the divide-by-2 stage.
- If the range of operating voltages for the divide-by-two and the strobe pulse generator overlap, you should be able to set the switch at INPUT 4 HI again, and observe the strobe pulses divided by two, at a suitably chosen V_{DD}. However, it is possible you will come to the conclusion that the operable ranges do not coincide. Full operation is only possible on a few sample chips where fabrication tolerances dictate. This idiosyncratic behaviour cannot be directly predicted by the simulation tools with the models available.

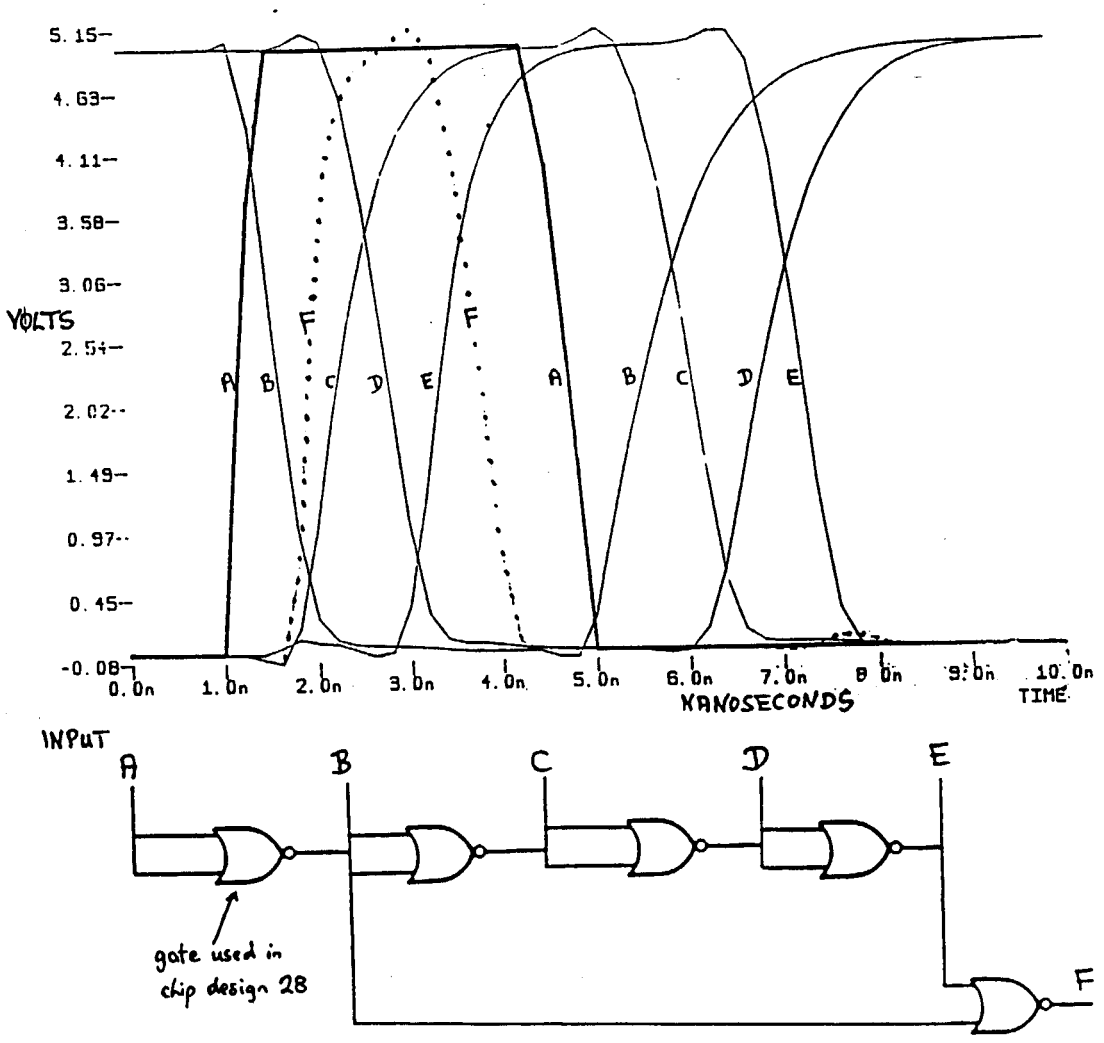


Figure 4 Simulation at V_{DD}=5V of a portion of ring 28